NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

RAISING THE DEGREE OF SERVICE-ORIENTATION OF A SOA-BASED SOFTWARE SYSTEM: A CASE STUDY

by

Feng Shi Liu

December 2009

Thesis Co-Advisors:	Man-Tak Shing Bret Michael

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE: Raising the Degree of Service-Orientation of a SOA-based Software System: A Case Study	**5. FUNDING NUMBERS**
6. AUTHOR(S) Feng Shi Liu	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

The term "service-oriented architecture (SOA)" has been gaining popularity in the Department of Defense's (DoD) software engineering and IT community in recent years. It has become the next "big thing" and is the buzz word used at technical conferences and high level management meetings. But the term "service-orientation" has caused much confusion among program sponsors, government IT managers, and software professionals. Its apparent ambiguity has let them to claim their own interpretations of SOA. Many have been led to the notion that a technical architecture deemed service-oriented is simply one comprised of Web services. This is a common but dangerous assumption that leads to the number one mistake made by projects intending to adopt SOA—the perception that the benefits promised by current mainstream SOA are attainable solely through an implementation using the Web services platform.

This paper will present a case study to illustrate that building an SOA-based application is not just about applying a particular set of technologies and standards but by following a set of sound design principles based on service-orientation. The biggest contribution of this paper is to show, by conducting a case study, that the use of Web services alone does not make a system service-oriented. The results of this paper can be used by IT professionals in the DoD to better evaluate the degree of service-orientation for a software system's architecture.

14. SUBJECT TERMS SOA, Web services, open architecture, Command and Control, Sensor Management	**15. NUMBER OF PAGES** 95
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

RAISING THE DEGREE OF SERVICE-ORIENTATION OF A SOA-BASED SOFTWARE SYSTEM: A CASE STUDY

Feng S. Liu
Civilian, United States Navy
B.S., University of Kansas, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 2009

Author: Feng S. Liu

Approved by: Man-Tak Shing
Thesis Advisor

Bret Michael
Co-Advisor

Peter J. Jenning
Chairman, Department of Software Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The term “service-oriented architecture (SOA)” has been gaining popularity in the Department of Defense’s (DoD) software engineering and IT community in recent years. It has become the next “big thing” and is the buzz word used at technical conferences and high level management meetings. But the term “service-orientation” has caused much confusion among program sponsors, government IT managers, and software professionals. Its apparent ambiguity has let them to claim their own interpretations of SOA. Many have been led to the notion that a technical architecture deemed service-oriented is simply one comprised of Web services. This is a common but dangerous assumption that leads to the number one mistake made by projects intending to adopt SOA—the perception that the benefits promised by current mainstream SOA are attainable solely through an implementation using the Web services platform.

This paper will present a case study to illustrate that building an SOA-based application is not just about applying a particular set of technologies and standards but by following a set of sound design principles based on service-orientation. The biggest contribution of this paper is to show, by conducting a case study, that the use of Web services alone does not make a system service-oriented. The results of this paper can be used by IT professionals in the DoD to better evaluate the degree of service-orientation for a software system’s architecture.

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This thesis is dedicated to my wife, Molly, and our beautiful princess, Joanne, who supported me while I was working late trying to finish this project.

I would also like to thank my advisor, Professor Shing, for the time and effort he has provided throughout this project. He has been a great teacher to me. I am a better software engineer today because of his guidance and outstanding mentorship.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

The Department of Defense (DoD) has singled out Service-Oriented Architecture (SOA) implementation as the best way to “fulfill the requirements of a net-centric environment” [1]. But the term SOA has been widely misunderstood within DoD’s software community. The major confusion comes from the fact that many software professionals cannot distinguish the difference between SOA, a software design concept, and Web services, a set of technologies and standards. They often use the two terms interchangeably.

This apparent ambiguity has let program sponsors, government managers, and software engineers to claim their own interpretations of SOA. Many have been led to the notion that a technical architecture deemed service-oriented is simply one comprised of Web services. This is a common but dangerous assumption that leads to the number one mistake made by projects intending to adopt SOA—the perception that the benefits promised by current mainstream SOA are attainable solely through an implementation using the Web services platform [2].

Throughout my software engineering career at the Space and Naval Warfare Systems Center Pacific in the last six years, I have encountered numerous applications that apply Web services based on a non-SOA approach. This is due to an improper understanding of the basic fundamental design principles of SOA. Web services and SOA, even though closely related, are fundamentally different. It’s possible to build an application with Web services without being service-oriented, and it’s also possible to build a SOA-based application without using Web services at all.

One can’t get the benefits that SOA offers unless one truly understands its basic underlying principles, and one can’t leverage the benefits that Web services provide unless Web services are applied based on SOA. Without knowing the proper relationship between the two, SOA and Web services will fail to fulfill their promises and realize their full potential.

B. PURPOSE

This thesis attempts to answer the following questions:

1) Does a system's use of Web services make its architecture service-oriented?
2) What determines whether a system is designed based on SOA?
3) What criteria can be used to evaluate a system's degree of service-orientation?

The most efficient way to answer the above questions is to conduct a detailed case study. Albert Einstein once said that "*example isn't another way to teach, it is the only way to teach.*" The use of a case study offers a different approach than many books and papers on the subject which put a heavy emphasis on theory and concepts instead of using detailed examples.

Thus, in this thesis, a detailed case study on the Sensor Management System/Joint Perimeter Surveillance Command Control Integrated System (SMS/JPSC2) will be conducted to address the above questions.

The thesis will present, through the study of SMS/JPSC2, that building an SOA-based application is not just about applying a particular set of technologies and standards but by following a set of sound design principles based on service-orientation. It describes an alternative architectural design and a set of new services for SMS to improve its degree of service-orientation. The biggest contribution of this thesis is to show that the use of Web services alone does not make a system service-oriented. SOA and Web services, though related, are fundamentally different. The results of this thesis can help IT professionals to gain a better understanding of SOA and Web services, their relationships, and how to evaluate a software system's architecture based on service-oriented design principles.

C. ORGANIZATION

This thesis is organized as follows:

- Chapter I is the introduction section of the thesis. It presents the motivation, purpose, and organization of this thesis.
- Chapter II provides background information on Service-Oriented Architecture (SOA), Web services technology and standards, and the case study on Sensor

Management System/Joint Perimeter Surveillance Command Control Integrated System (SMS/JPSC2).

- Chapter III describes the current architectural design of the Sensor Management System (SMS).
- Chapter IV presents the analysis and evaluation of SMS’s architecture based on SOA design principles.
- Chapter V presents an alternative architectural design and a set of new services for SMS to improve its degree of service-orientation.
- Chapter VI summarizes the thesis, and makes suggestions for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

A. SERVICE-ORIENTED ARCHITECTURE

This chapter provides background information about SOA for readers new to the subject so that they can put the material presented in the remaining chapters into the proper context.

Before we define what a Service-oriented Architecture is, let's first define what a service is under the context of SOA:

> A service is an implementation of a well-defined piece of business functionality, with a published interface that is discoverable and can be used by service consumers when building different applications and business processes [3].

With the term 'service' defined, let's attempt to define SOA:

> Service-Oriented Architecture is a software design methodology that uses loosely-coupled services to perform business functions or processes. These services communicate using well-defined standards [3].

The Net-Centric Enterprise Solutions for Interoperability (NESI) overview document explains SOA as follows:

> SOA promotes flexibility and reuse. This enables developers to compose complex software systems from clearly defined, implementation-neutral interfaces rather than through brittle implementation mechanisms such as tightly coupled, highly integrated applications… SOA isolates the specifics of data implementation from the service interface… Services are designed to be highly interoperable, loosely coupled, decentralized, and discoverable across the enterprise [5].

In the world of SOA, applications govern their individual services; each service evolves and grows relatively independent from each other. However, in order for those independent and autonomous services to work seamlessly together, they need to adhere to certain baseline conventions. These conventions standardize key aspects of each business for the benefit of the service consumers without adversely affecting individual application's ability to exercise self-governess [2].

Processing in SOA is highly distributed. Each service has an explicit functional boundary and related resource requirements. In modeling a technical service-oriented architecture, we have many choices as to how we can position and deploy services. Enterprise solutions consist of multiple servers, each hosting sets of Web services and supporting middleware. Services can be distributed as required, and performance demands are one of several factors in determining the physical deployment configuration [2].

Here's a good analogy: a SOA-based service is like a tangram puzzle. Tangram pieces are "loosely coupled" and provide the flexibility to create a wide variety of products using the same pieces. This is very illustrative of composing SOA services to serve multiple business processes. Traditional applications based on component architecture were built to satisfy one business process. The tangram puzzle or service "modules" are constructed with loosely-coupled interfaces to allow for business process flexibility and use in multiple business processes. When properly designed, loosely coupled services support a composition model, allowing individual services to participate in aggregate assemblies. This introduces continual opportunities for reuse and extensibility.

1. Fundamental Service-oriented Design Principles

SOA can also be viewed as a form of technology architecture that adheres to the principles of service-orientation.

That definition begs the question: what are the principles of service-orientation? Thomas Erl, a world renowned expert on SOA, defined service oriented design principles as follows:

Services are reusable. Logic is divided into services with the intention of promoting reuse. Regardless of whether immediate reuse opportunities exist, services are designed to support potential reuse. By applying design standards that make each service potentially reusable, the chances of being able to accommodate future requirements with less development effort are increased [2].

Services share a formal contract. Service contracts provide a formal definition of:

- service endpoint
- each service operation
- every input and output message supported by each operation
- rules and characteristics of the service and its operations.

Service contracts therefore define almost all of the primary parts of an SOA. Good service contracts also may provide semantic information that explains how a service may go about accomplishing a particular task. Either way, this information establishes the agreement made by a service provider and service requestors [2].

Services are loosely coupled. Services maintain a relationship that minimizes dependencies and only requires that they retain an awareness of each other. They must be designed to interact on a loosely coupled basis, and they must maintain this state of loose coupling. This is closely related to service abstraction and service autonomy. [*Loosely coupled frameworks allow individual nodes in a distributed system to change without affecting or requiring change in any other part of the system.*]

Being able to ultimately respond to unforeseen changes in an efficient manner is a key goal of applying service-orientation. Realizing this form of agility is directly supported by establishing a loosely coupled relationship between services [2]. Very loosely coupled systems have the added advantage that they tend to have shorter development time. This is due to the low amounts of inter-module dependency.

Services abstract underlying logic. Beyond what is described in the service contract, services hide logic from the outside world. The only part of a service that is visible to the outside world is what is exposed via the service's description and formal contract. The underlying logic is invisible and irrelevant to service requestors.

Services are composable. Collections of services can be coordinated and assembled to form composite services. This possibility allows logic to be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.

Services are autonomous. Services have control over the logic they encapsulate. The logic governed by a service resides within an explicit boundary. The service has complete autonomy within this boundary and is not dependent on other services for the execution of this governance. It also eliminates dependencies on other services, which frees a service from ties that could inhibit its deployment and evolution [2].

Services are stateless. Services minimize retaining information specific to an activity. They should not be required to manage state information, since that can impede their ability to remain loosely coupled. Stateless is a preferred condition for services and one that promotes reusability and scalability.

Services are discoverable. Discovery helps avoid the accidental creation of redundant services or services that implement redundant logic. Because each operation provides a potentially reusable piece of processing logic, metadata attached to a service needs to sufficiently describe not only the service's overall purpose, but also the functionality offered by its operations [2].

Thus services should be designed to be outwardly descriptive so that they can be found and assessed via availability discovery mechanisms. They should allow their descriptions to be discovered and understood by humans and service users who may be able to make use of the services' logic. Service discovery can be facilitated by the use of a directory provider such as the UDDI registry [6].

In addition, we added one more SOA design principle to the list:

Services are modular. Modularity represents a distinct approach for separating concerns. What this means is that logic required to solve a large problem can be better constructed, carried out, and managed if it is decomposed into a collection of smaller, related pieces [2]. Each of these pieces addresses a concern or a specific part of the problem. The concept of modularity is nothing new. It's an old design concept promoted in many traditional architectural approach. What distinguishes SOA from them is that SOA services are autonomous and loosely-coupled. A good analogy is to think of component-based architecture as jigsaw puzzles (tightly coupled) and SOA-based architecture as tangram puzzles (loosely coupled).

2. Pitfalls of SOA Design

Fundamental service-orientation principles are designed to be technology agnostic. Building applications with service-oriented architecture requires a sound understanding of basic software design principles specified above.

The list below identifies some of the common pitfalls of designing SOA-based applications:

- Improper partitioning of functional boundaries within services
- Creation of non-composable (or semi-composable) services
- Creation of tightly coupled services
- Creation of stateful Web services

In Chapter IV, we will apply the above principles to evaluate SMS/JPSC2's system architecture to determine its degree of service-orientation and to propose alternative design solutions to make it more service-oriented.

B. WEB SERVICES TECHNOLOGY AND STANDARDS

A Web service is defined by the World Wide Web Consortium (W3C) as the following:

> A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. [2]

To put it simply, Web services encompass a set of related standards that can enable any two computer applications to communicate and exchange data via the common Internet protocols.

One key benefit that Web services offer that traditional distributed architectures do not is that Web services are based on open standards.

We shall look at Web services standards in the next section.

1. Web Services Standards

The core set of Web services standards includes the followings:

- Extensible Markup Language (XML) [7]
- Web Service Description Language (WSDL) [8]
- Simple Object Access Protocol (SOAP) [9]
- Universal Description Discovery and Integration **(**UDDI)

a. Extensible Markup Language (XML)

XML is the main standard used in Web services. It is a language for marking up data so that information can be exchanged between applications and platforms.

b. Simple Object Access Protocol (SOAP)

SOAP is a messaging protocol for transporting information and instructions between Web services, using XML as a foundation for the protocol. It also defines a way to perform remote procedure calls (RPCs) using Hypertext Transfer Protocol (HTTP) as the underlying communication protocol.

c. Web Service Description Language (WSDL)

Web services Description Language provides a standard method of describing Web services and their specific capabilities. WSDL is a XML-based language used to describe what a Web service can do, where it resides, and how to invoke it. The WSDL serves as contract between the Web service and a consumer or potential consumer of that service. The WSDL file describes both the data to be passed and the method for passing the data.

d. Universal Description Discovery and Integration (UDDI)

Universal Description, Discovery and Integration defines XML-based rule for building directories in which applications advertise themselves and their Web services. It also provides an interface and a mechanism for clients to dynamically find Web services offered by external applications. A UDDI registry has two kinds of clients: applications that want to publish a service (and its usage interfaces), and clients who want to obtain services of a certain kind and bind programmatically to them.

UDDI can also be viewed as a registry of descriptions of Web services available for use much like telephone yellow pages provides information about available commercial services. The registry itself is a hierarchical structure of business, service, and binding information represented in XML. The purpose of UDDI is to make service discovery possible at design time and dynamically at runtime.

2. Web Service Stack

The Web services stack shows the collection of computer networking protocols that define, locate, implement, and make Web services interact with each other. The World Wide Web Consortium's Web Services Architecture Working Group defined technical standards to ensure interoperability for SOAs.

The Working Group divided these standards into the following six areas: processes, descriptions, messages, communications, security and management: Figure 1 shows a modified version of their Web Services Architecture Stack diagram.

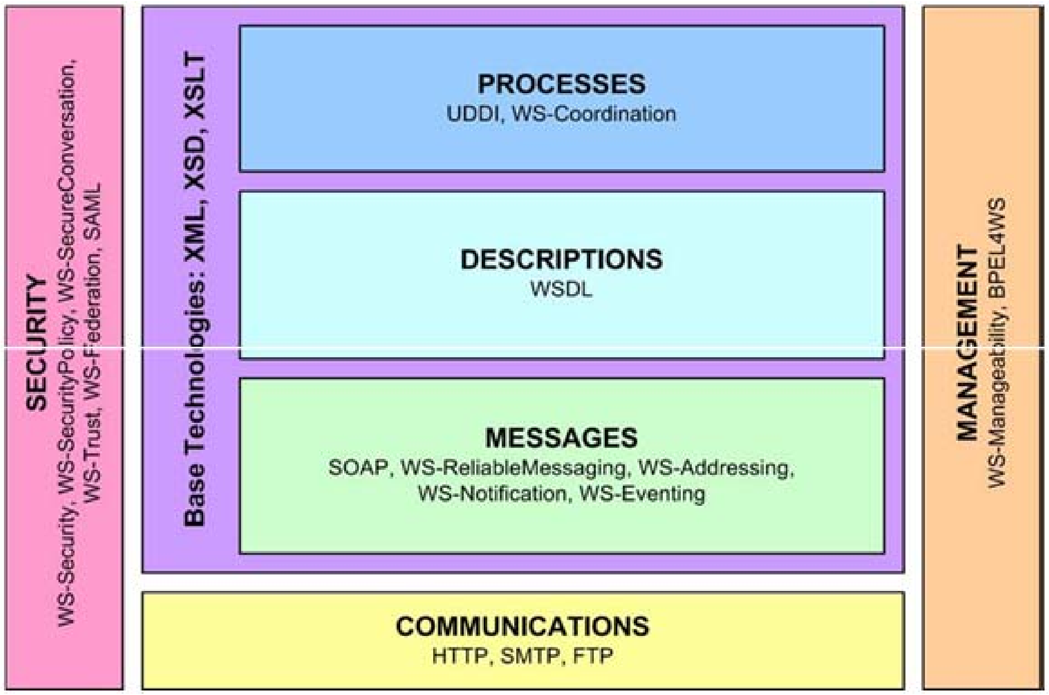

SECURITY
WS-Security, WS-SecurityPolicy, WS-SecureConversation, WS-Trust, WS-Federation, SAML
Base Technologies: XML, XSD, XSLT
PROCESSES
UDDI, WS-Coordination
DESCRIPTIONS
WSDL
MESSAGES
SOAP, WS-ReliableMessaging, WS-Addressing, WS-Notification, WS-Eventing
COMMUNICATIONS
HTTP, SMTP, FTP
MANAGEMENT
WS-Manageability, BPEL4WS

Figure 1: Web Services Architecture Stack Diagram

a. Process Layer

The Process layer describes how providers publish services and requestors/consumers discover them. The Process layer utilizes the following standards:

- *Universal Description Discovery and Integration (UDDI)*: Again as mentioned earlier, UDDI is a directory that allows applications to register their Web services so that the potential service consumers can find them.
- *WS-Coordination*: This specification "describes an extensible framework for providing protocols that coordinate the actions of distributed applications. Such coordination protocols are used to support a number of applications, including those that need to reach consistent agreement on the outcome of distributed activities" [10].

b. Description Layer

The Description layer describes how the service provider communicates the specifications for invoking the Web service to the service requestor. The Description layer utilizes the following standards:

- *Web Service Description Language (WSDL)*: An XML document that describes the interfaces and methods that a service provides.

c. Messages Layer

The Messages layer describes how the services pass information in the form of a message. The Messages layer utilizes the following standards:

- *Simple Object Access Protocol (SOAP)*: SOAP is a protocol used to exchange messages between systems in XML format. SOAP has become the de-facto standard protocol for Web services [9].
- *WS-ReliableMessaging*: This specification describes a protocol that allows messages to be transferred reliably between nodes in the presence of software component, system, or network failures [31].
- *WS-Addressing*: This specification "provides transport-neutral mechanisms to address Web services and messages. Specifically, this specification defines XML elements to identify Web service endpoints and to secure end-to-end endpoint identification in messages. This specification enables messaging systems to support message transmission through networks that include processing nodes such as endpoint managers, firewalls, and gateways in a transport-neutral manner" [11].
- *WS-Notification*: "The Event-driven, or Notification-based, interaction pattern is a commonly used pattern for inter-object communications. Examples exist in many domains, for example in publish/subscribe systems provided by Message Oriented Middleware vendors, or in system and device management domains" [12].

- *WS-Eventing*: "This specification describes a protocol that allows Web services to subscribe to or accept subscriptions for event notification messages" [13].

d. Communications Layer

The Communications layer describes how messages are physically transported across the network. The Communications layer utilizes the following Internet protocols:

- *Hypertext Transfer Protocol (HTTP)*: HTTP is the standard mechanism for retrieving Web pages and associated content. It can also be used for transmitting data from the client to the server [14].
- *Simple Mail Transfer Protocol (SMTP)*: SMTP is the standard mechanism for sending email from the client to the server [15].
- *File Transfer Protocol (FTP)*: FTP is primarily used for transferring files from one computer to another over a TCP/IP network [16].

e. Security

Security occurs at all layers in the stack and it provides authenticity, integrity, confidentiality, and non-repudiation. Security utilizes the following standards:

- *WS-Security*: "This specification describes enhancements to SOAP messaging to provide message integrity and confidentiality. The specified mechanisms can be used to accommodate a wide variety of security models and encryption technologies" [17].
- *WS-SecurityPolicy*: WS-SecurityPolicy is designed to work with the general Web Services framework including WSDL service descriptions, UDDI businessServices and bindingTemplates, and SOAP message structure and message processing model. WS-SecurityPolicy should be applicable to any version of SOAP [18].

- *WS-SecureConversation*: "This specification defines extensions that build on WS-Security to provide a framework for requesting and issuing security tokens, and to broker trust relationships" [19].
- *WS-Trust*: The goal of WS-Trust is to enable applications to construct trusted SOAP message exchanges. This trust is represented through the exchange and brokering of security tokens. This specification provides a protocol agnostic way to issue, renew, and validate these security tokens [20].
- *WS-Federation*: A specification, by IBM and Microsoft, for standardizing the way companies share user and machine identities among disparate authentication and authorization systems spread across corporate boundaries [21].
- *SAML*: "An XML-based framework for communicating user authentication, entitlement, and attribute information. As its name suggests, SAML allows business entities to make assertions regarding the identity, attributes, and entitlements of a subject (an entity that is often a human user) to other entities, such as a partner company or another enterprise application" [22].

f. Management

Management, like Security, occurs across all layers in the stack. Management provides methods for monitoring and managing services and business processes. Management utilizes the following standards:

- *WS-Manageability*: "specification introduces the general concepts of a manageability model in terms of manageability topics and the aspects used to define them" [23].
- *Business Process Execution Language for Web Services* (*BPEL4WS*): "The Business Process Execution Language for Web Services provides a comprehensive syntax for describing business workflow logic. It allows for the creation of abstract processes that can describe business

protocols, as well as executable processes that can be compiled into runtime scripts” [2] The Business Process Modeling Notation (BPMN) provides a standardized graphical notation for drawing business processes in a workflow. Software tools easily translate BMPN models into BPEL4WS files [24].

3. Relationship between SOA and Web Services

Thomas Erl, in his book *Service-Oriented Architecture: Concepts, Technology, and Design* [2], coined the term Contemporary SOA, which can be defined as an extended variation of the primitive Service-oriented Architecture we defined in the last section. It has the following characteristics:

- Contemporary SOA increases quality of service
- Contemporary SOA is fundamentally autonomous
- Contemporary SOA is based on open standards
- Contemporary SOA supports vendor diversity
- Contemporary SOA fosters intrinsic interoperability
- Contemporary SOA promotes discovery
- Contemporary SOA promotes federation
- Contemporary SOA promotes architectural composability
- Contemporary SOA fosters inherent reusability
- Contemporary SOA emphasizes extensibility
- Contemporary SOA supports a service-oriented business modeling paradigm
- Contemporary SOA implements layers of abstraction
- Contemporary SOA promotes loose coupling throughout the enterprise
- Contemporary SOA promotes organization agility

According to Erl, the relationship between SOA and Web services can be defined as such:

Contemporary SOA represents an architecture that promotes service-orientation through the use of Web services.

SOA is a concept, an abstract idea, and a set of design principles, whereas Web services are a set of technologies and standards that, if utilized correctly, will facilitate the process of building SOA-based applications. In other words, if SOA represents the ideal of building reusable, agile, interoperable, and loosely-coupled software, then Web services represent a means to achieve it.

Web service provides an open, standardized interface. This interface supports the open communications framework that sits at the core of Contemporary SOA and establishes an environment under which building loosely coupled software services are promoted and simplified.

Thus, to realize the full potential benefits of SOA, software designers need to standardize how Web services are positioned and designed, according to service-orientation principles.

A technical and conceptual knowledge of Web services is certainly helpful. However, as we established at the beginning of this chapter, fundamental service-orientation principles are technology agnostic. Building applications based on service-oriented architecture requires a sound understanding of basic software design principles. The emphasis placed on business logic encapsulation and the creation of service abstraction layers often will require a blend of technology, business analysis expertise, and software design best practices. It is best to assume that realizing contemporary SOA requires a set of skills that goes beyond the knowledge of Web services technology.

In this thesis, by using a case study, we will attempt to illustrate that the application of Web services alone is not sufficient in building SOA-based software applications.

C. CASE STUDY: SMS/JPSC2

This section provides a very high level view of our case study—the Sensor Management System/Joint Perimeter Surveillance Command Control Integrated System (SMS/JPSC2).

1. System Overview

SMS/JPSC2 is a multi-purpose surveillance system designed to provide perimeter surveillance to a designated geographical area (Figure 2). The system is sponsored by the Commander Naval Installations (CNI), and developed by the Space and Warfare Systems Center (SPAWAR) Code 2644 in San Diego, CA. The system has already been installed and operational at the U.S. Coast Guard's facility at San Diego, CA; Seattle, WA (USCG/USN); Jacksonville, FL (USN/USCG); and at the Lemoore Naval Air Base, CA.

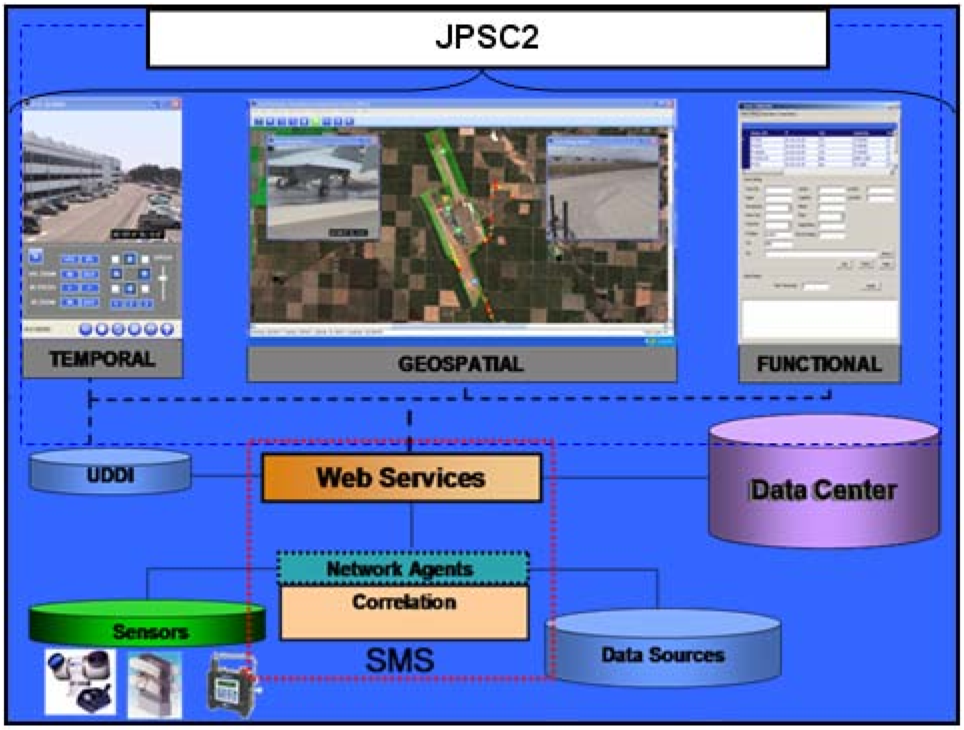


Figure 2: Basic SMS/JPSC2 Schematic

SMS/JPSC2 has two major subsystems: the Joint Perimeter Surveillance Command and Control (JPSC2) Integrated System and the Sensor Management System (SMS). In this section, we will introduce the major functionalities and architectural design of the two major subsystems.

2. JPSC2

JPSC2 provides a user graphical interface for communicating and controlling surveillance assets. Surveillance assets can be defined as sensors (radars, cameras, transponders, video detection devices, etc.) that have the capabilities to detect, track, and report targets of interests. JPSC2 communicates with surveillance sensors via the Sensor Management System (which will be covered in detail in later sections) and present the collected sensor data on a geographical map. When JPSC2 presents detected objects on the viewing screen (called "tracks"), it enables operators to select the tracks to view their detailed information such as name, location, heading, speed, etc. JPSC2 also integrates live surveillance camera video feeds and display them to the Watch Security Officer through its graphical user interface. (See Figure 3 for a snapshot of JPSC2.) Surveillance cameras can be controlled by the Watch Security Officer through the JPSC2 interface. Camera control functions include slewing cameras to specific targets and setting cameras to follow a selected target.

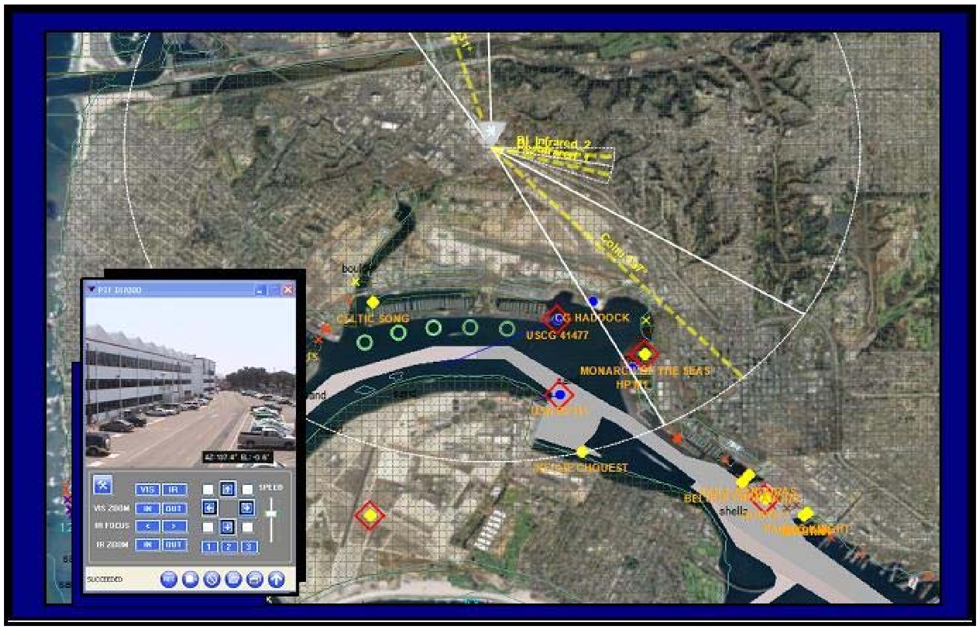


Figure 3: JPSC2 User Interface: Regional Surveillance

Below is a partial list of the basic command and control functionalities that JSPC2 provides:

- Real-time Detection of Objects—Within established alarm zones, JSPC2 automatically monitors the defined perimeters based on a set of pre-defined rules and alerts operators of any violations. Upon detection of a violation, the system automatically slews the nearest camera to the violating target and tracks the intruder continuously, providing the Security Watch Offer the exact location to which a reaction force can be directed.
- Display near real-time tracks collected by remote sensors on geographical maps of surveillance areas
- Monitor, identify, and track targets by directing and controlling remote sensors
- Operate on and make technical adjustments to local and remote sensor equipments such as cameras, ground and marine radars
- View/Query identification data transmitted from land and marine vessels
- View live tracks; store and retrieve historical track data
- View live video feeds from one or more remote cameras
- Record, store, and review snapshots and brief video clips
- Establish stationary and moving alarm zones based on a set of pre-defined business rules
- Compile, review, sort, and prioritize alarms and incidents
- Enable/disable audible detection alarms and enable/disable a predefined schedule of detection alarms

3. Sensor Management System

SMS aggregates data collected from multiple surveillance sensors and make those data accessible for client systems such as JPSC2 via the Sensor Data Web service interface of its Web Server Process (Figure 5). SMS also allows client C2 systems to direct and control surveillance sensors via the Sensor Control Web service interface of its Web Server Process. The basic design goal for SMS is to decouple the management of sensors from specific command and control systems (C2s). SMS provides all the "plumbing" required to establish connections to the sensors, receive messages from them, and translates each sensor's proprietary message format into a common data format. It basically provides a layer of abstraction between client C2 systems and the surveillance sensors that the C2 system wants to communicate. A detailed discussion on SMS architecture will be presented in the next chapter.

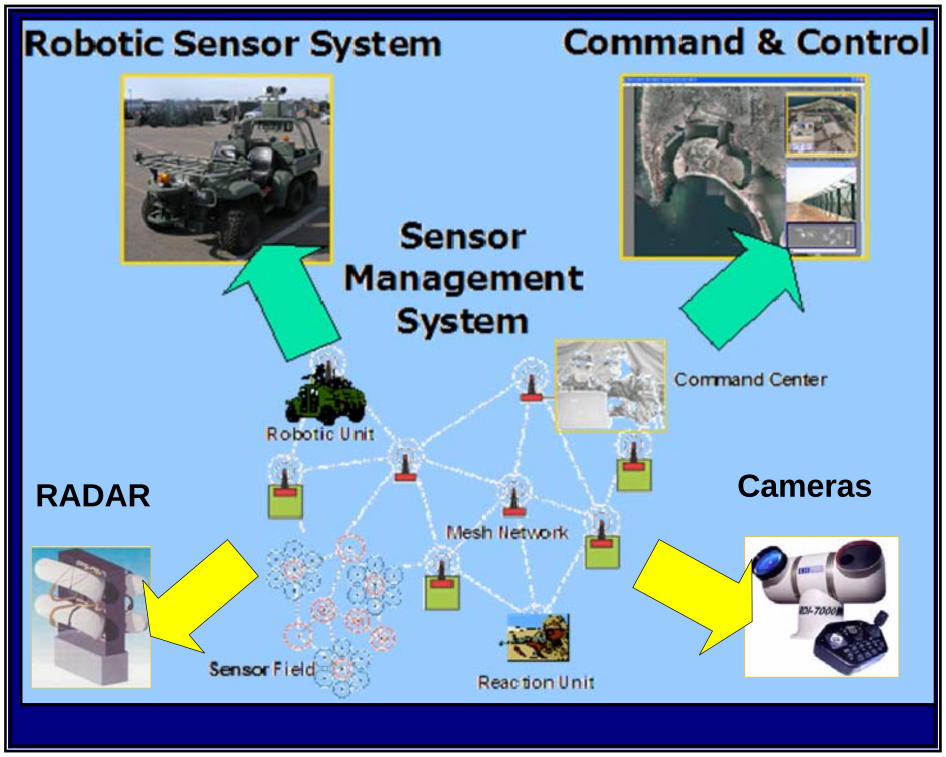


Figure 4: Sensor Management System Overview

4. System Architecture

Figure 5 shows the high level architectural components of SMS/JPSC2.

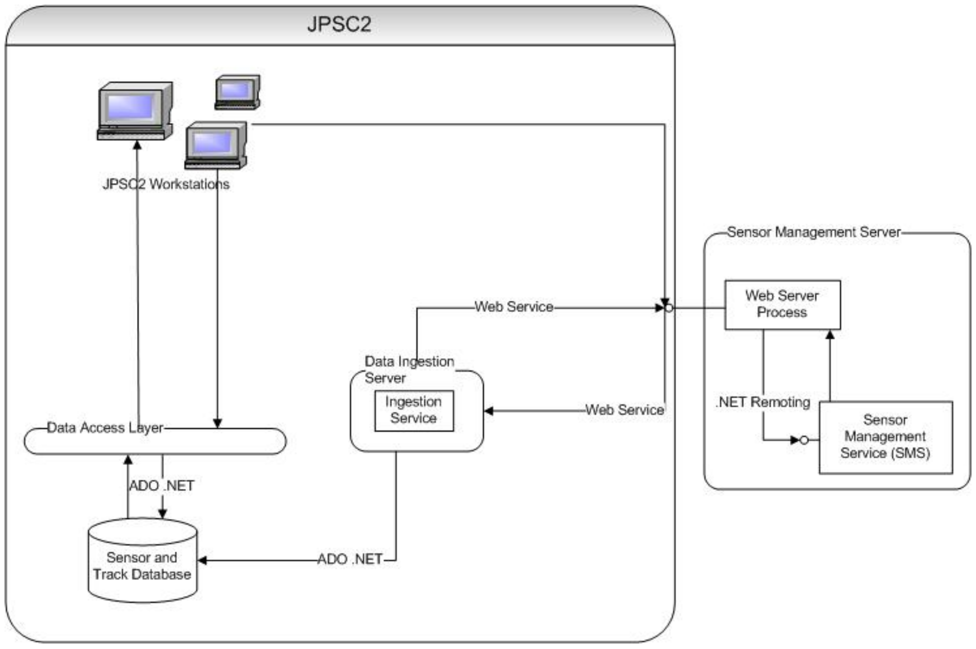


Figure 5: Current System Architecture of SMS/JPSC2

As Figure 5 shows SMS is the backend server process that aggregates data collected by various surveillance sensors and makes them available to client systems through its Sensor Data Web service interface. An intermediary service in JPSC2 - the Data Ingestion Service, retrieves the collected sensor data via SMS's Sensor Data Web service interface and stores them into the JPSC2 Data Center. The Data Ingestion Service is a multi-threaded application that can receive sensor data feeds from multiple SMS servers.

The JPSC2 Data Center provides data storage to both live and historical tracks. Virtually every activity detected by the surveillance sensors managed by SMS can be stored to and retrieved from the Data Center.

Once sensor data are stored in the JPSC2 Data Center, they are available for JPSC2 clients to retrieve for display on a Command and Control map console. A JPSC2 client is basically a thick client running on some user workstation. It is the presentation layer of the system. Most of the graphical user interface related programming logic is implemented within JPSC2 client tier.

JPSC2 can also communicate with surveillance sensors through SMS's Sensor Control Web service interface. This takes place when the operator tries to control a surveillance sensor such as configuring radar settings or moving cameras. Upon receiving a control command from JPSC2, SMS translates the command to a sensor specific message format and forward the command to the appropriate sensor using the sensor's communication protocol.

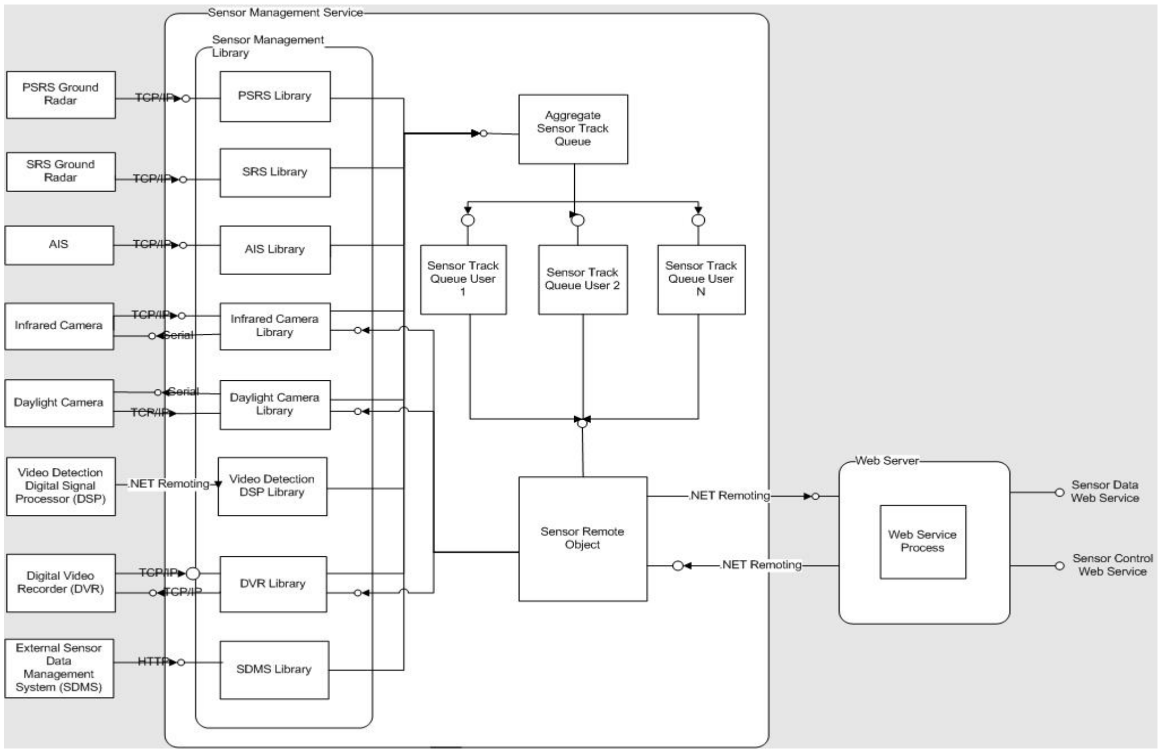
THIS PAGE INTENTIONALLY LEFT BLANK

III. CURRENT DESGIN OF SMS

A. INTERFACE TO SENSOR SYSTEMS

In this section, we will present how SMS works at the architectural level in general and how it communicates with external systems in particular.

SMS provides the communication backend to integrate and control sensors. Sensors in the context of SMS can be defined as any device that can detect objects-of-interests and report their status and positional information to a client system in near real-time. Some of the sensors that SMS manages also have remote interfaces that accept control commands from client systems. Examples of SMS sensors include ground radars, marine radars, Automatic Identification System (AIS) transponders, video detection digital signal processors, and surveillance cameras. Figure 6 shows the major architectural components of SMS.



Sensor Management Service
Sensor Management Library
PSRS Ground Radar
TCP/IP
PSRS Library
SRS Ground Radar
SRS Library
AIS
AIS Library
Infrared Camera
Serial
Infrared Camera Library
Daylight Camera
Daylight Camera Library
Video Detection Digital Signal Processor (DSP)
.NET Remoting
Video Detection DSP Library
Digital Video Recorder (DVR)
DVR Library
External Sensor Data Management System (SDMS)
HTTP
SDMS Library
Aggregate Sensor Track Queue
Sensor Track Queue User 1
Sensor Track Queue User 2
Sensor Track Queue User N
Sensor Remote Object
Web Server
Web Service Process
Sensor Data Web Service
Sensor Control Web Service

Figure 6: SMS Architectural Component Diagram

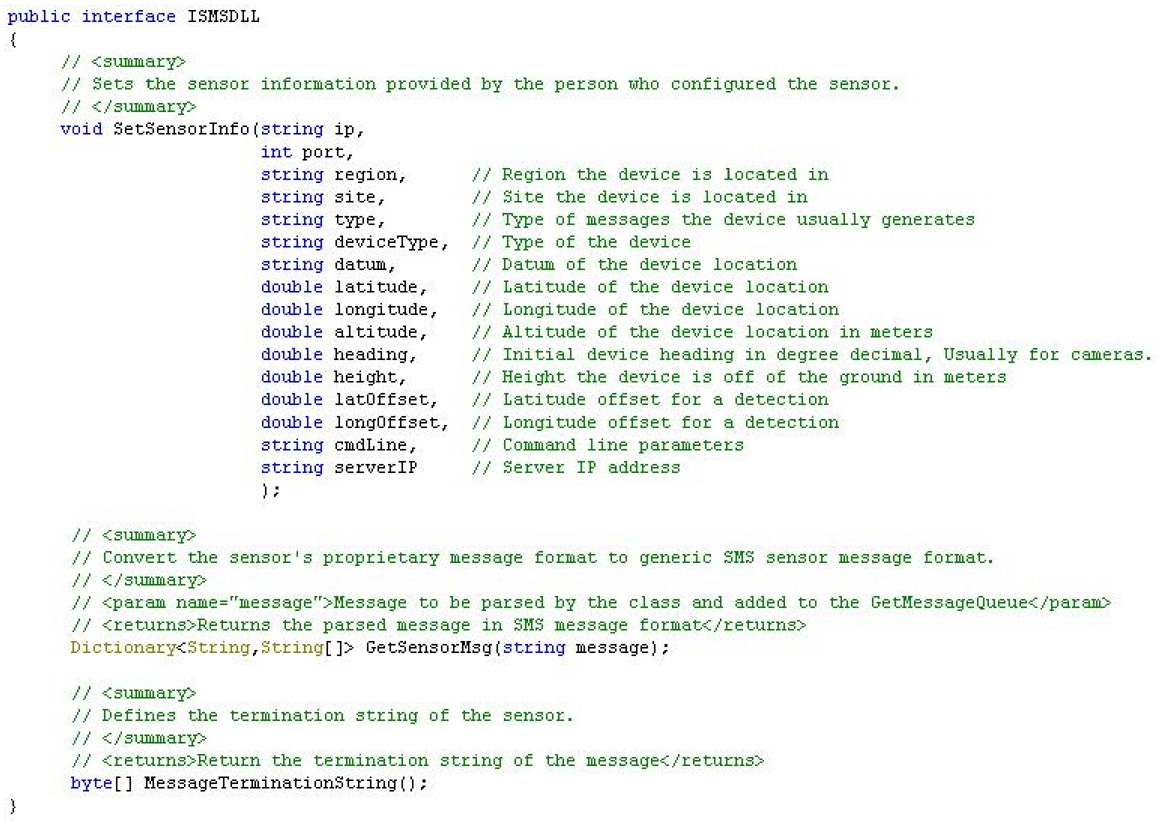
Now, let’s take a look at how surveillance sensors are managed by SMS.

SMS is essentially a Windows service implemented in Microsoft’s .NET framework. In SMS, there is a software library for every sensor that SMS manages. For example, there is a software library for communicating with AIS, a library to communicate with Perimeter Surveillance Radar system (PSRS), and a library to control surveillance cameras, etc. Since each sensor managed by SMS has its own communication protocol and messaging format, the corresponding sensor library has to be developed based on that sensor’s specific communications interface requirements. Table 1 shows a data exchange matrix between SMS and some of its managed sensors.

Table 1: SMS and External Sensor Systems Data Exchange Matrix

External Sensor System	Communication Protocol	Message Format	Data Flow	
			Source	Destination
AIS	TCP/IP	NMEA-0813	AIS	SMS
Surveillance Cameras	Serial	ASCII	SMS	Surveillance Cameras
PSRS ground radar	Serial	XML	PSRS	SMS
SRS ground radar	TCP/IP	XML	SRS	SMS
CamSmartz video detection digital signal processor (DSP)	TCP/IP	XML	CamSmartz	SMS
Vistascape Sensor Data Management System (SDMS)	HTTP	HTTP GET	SDMS	SMS
Mutiple-Input Tracking and Control System (MTRACS)	TCP/IP	OTH-GOLD	MTRACS	SMS
Multiple-Input Tracking and Control System (MTRACS)	TCP/IP	OTH-GOLD	SMS	MTRACS

SMS sensor libraries can be viewed as independent software components whose run-time behaviors are governed by SMS. When SMS is started, each active sensor library is loaded into SMS's process space as independent threads. They act as listeners listening either to incoming track reports from sensors or to sensor control commands triggered from command and control (C2) systems such as JPSC2. All sensor libraries implement the same interface. Figure 7 shows the interface that all SMS sensor libraries implement and provides descriptions for each interface method.

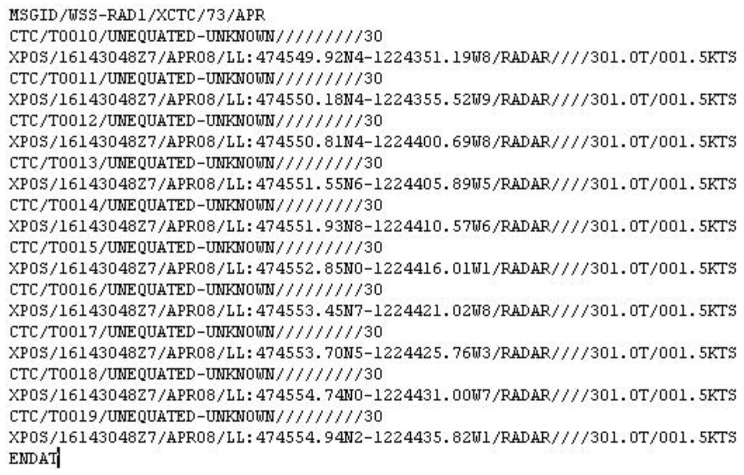


```
public interface ISMSDLL
{
     // <summary>
     // Sets the sensor information provided by the person who configured the sensor.
     // </summary>
     void SetSensorInfo(string ip,
                        int port,
                        string region,      // Region the device is located in
                        string site,        // Site the device is located in
                        string type,        // Type of messages the device usually generates
                        string deviceType,  // Type of the device
                        string datum,       // Datum of the device location
                        double latitude,    // Latitude of the device location
                        double longitude,   // Longitude of the device location
                        double altitude,    // Altitude of the device location in meters
                        double heading,     // Initial device heading in degree decimal, Usually for cameras.
                        double height,      // Height the device is off of the ground in meters
                        double latOffset,   // Latitude offset for a detection
                        double longOffset,  // Longitude offset for a detection
                        string cmdLine,     // Command line parameters
                        string serverIP     // Server IP address
                        );

      // <summary>
      // Convert the sensor's proprietary message format to generic SMS sensor message format.
      // </summary>
      // <param name="message">Message to be parsed by the class and added to the GetMessageQueue</param>
      // <returns>Returns the parsed message in SMS message format</returns>
      Dictionary<String,String[]> GetSensorMsg(string message);

      // <summary>
      // Defines the termination string of the sensor.
      // </summary>
      // <returns>Return the termination string of the message</returns>
      byte[] MessageTerminationString();
}
```

Figure 7: Sensor Interface

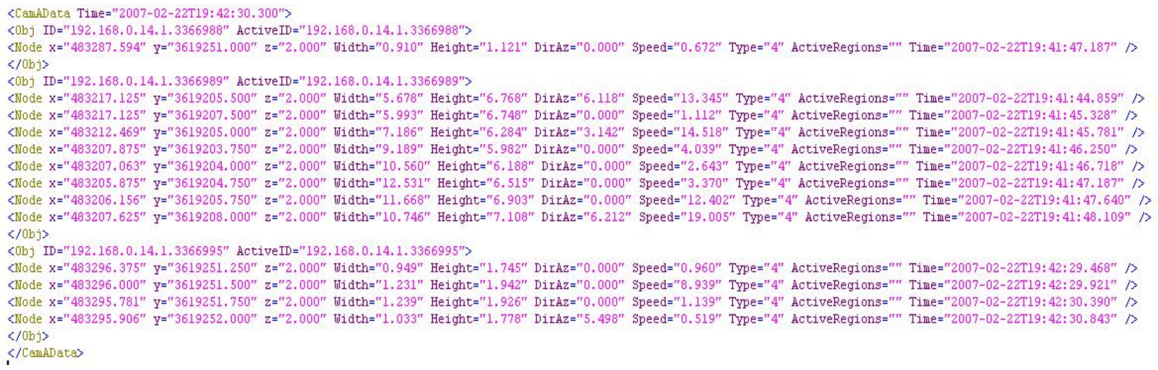
The *SetSensorInfo* method sets some sensor properties. Those properties usually come from the person who configures the sensor. The *GetSensorMsg* method converts the sensor's unique proprietary message format into SMS's generic sensor format. And finally the *MessageTerminationString* method defines the sensor message's termination string. Each sensor library is implemented as a .NET dynamic link library (dll). SMS has infrastructure services to load and run the sensor libraries using .NET Reflection by calling the three methods defined in the sensor interface. The details of how the SMS infrastructure services interact with SMS sensor libraries are beyond the scope of this thesis; we will not examine it further.

When SMS receives track reports from a given sensor, the corresponding sensor library in SMS converts those track reports from the sensor's own unique proprietary message format to a generic message format defined by SMS. We can view each sensor library as a translator that translates sensor specific "languages" into a "language" that SMS understands. Thus SMS provides the capability to merge sensor data gathered from multiple disparate data sources into a common messaging and communication protocol. Figures 8 and 9 show some sample proprietary messages from sensor systems, and Figure 10 shows a sample message in SMS format.



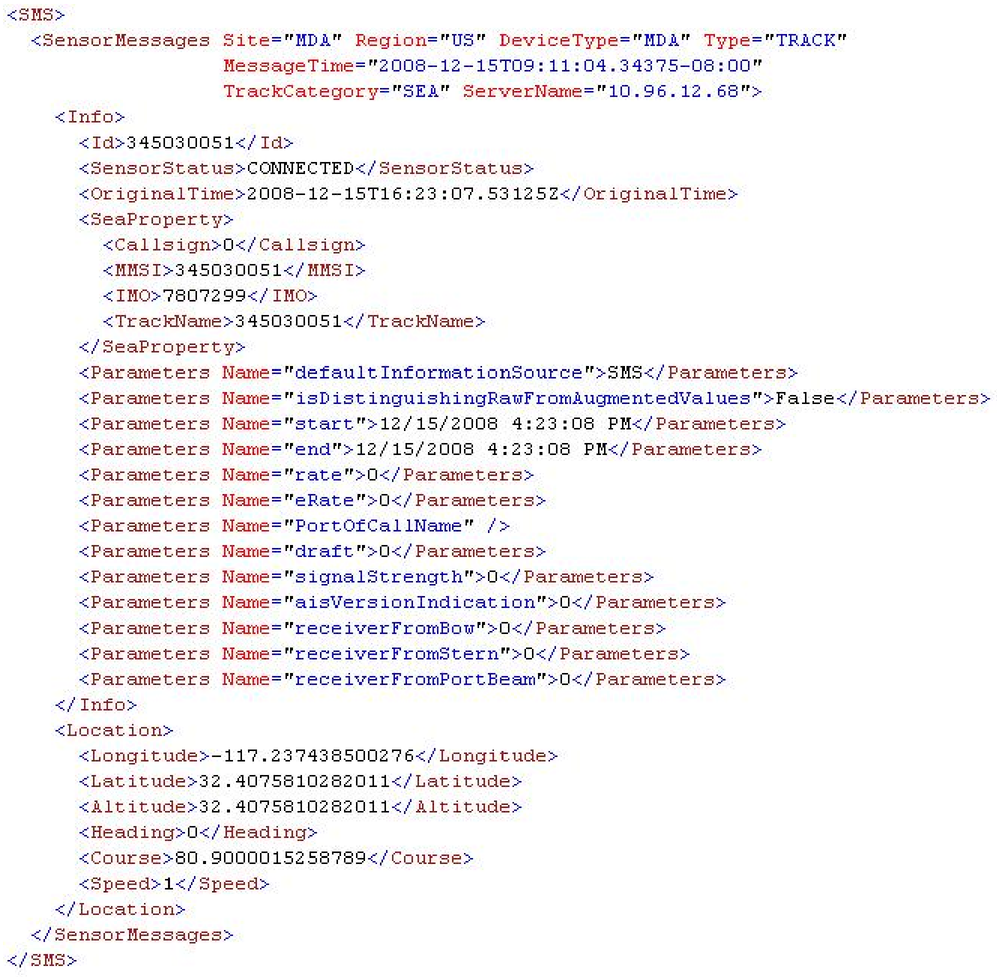
```
MSGID/WSS-RAD1/XCTC/73/APR
CTC/T0010/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474549.92N4-1224351.19W8/RADAR////301.0T/001.5KTS
CTC/T0011/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474550.18N4-1224355.52W9/RADAR////301.0T/001.5KTS
CTC/T0012/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474550.81N4-1224400.69W8/RADAR////301.0T/001.5KTS
CTC/T0013/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474551.55N6-1224405.89W5/RADAR////301.0T/001.5KTS
CTC/T0014/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474551.93N8-1224410.57W6/RADAR////301.0T/001.5KTS
CTC/T0015/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474552.85N0-1224416.01W1/RADAR////301.0T/001.5KTS
CTC/T0016/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474553.45N7-1224421.02W8/RADAR////301.0T/001.5KTS
CTC/T0017/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474553.70N5-1224425.76W3/RADAR////301.0T/001.5KTS
CTC/T0018/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474554.74N0-1224431.00W7/RADAR////301.0T/001.5KTS
CTC/T0019/UNEQUATED-UNKNOWN/////////30
XPOS/161430482Z7/APR08/LL:474554.94N2-1224435.82W1/RADAR////301.0T/001.5KTS
ENDAT
```

Figure 8: OTH-GOLD Message from GCCS-M



```
<CamAData Time="2007-02-22T19:42:30.300">
<Obj ID="192.168.0.14.1.3366988" ActiveID="192.168.0.14.1.3366988">
<Node x="483287.594" y="3619251.000" z="2.000" Width="0.910" Height="1.121" DirAz="0.000" Speed="0.672" Type="4" ActiveRegions="" Time="2007-02-22T19:41:47.187" />
</Obj>
<Obj ID="192.168.0.14.1.3366989" ActiveID="192.168.0.14.1.3366989">
<Node x="483217.125" y="3619205.500" z="2.000" Width="5.678" Height="6.768" DirAz="6.118" Speed="13.345" Type="4" ActiveRegions="" Time="2007-02-22T19:41:44.859" />
<Node x="483217.125" y="3619207.500" z="2.000" Width="5.993" Height="6.748" DirAz="0.000" Speed="1.112" Type="4" ActiveRegions="" Time="2007-02-22T19:41:45.328" />
<Node x="483212.469" y="3619205.000" z="2.000" Width="7.186" Height="6.284" DirAz="3.142" Speed="14.518" Type="4" ActiveRegions="" Time="2007-02-22T19:41:45.781" />
<Node x="483207.875" y="3619203.750" z="2.000" Width="9.189" Height="5.982" DirAz="0.000" Speed="4.039" Type="4" ActiveRegions="" Time="2007-02-22T19:41:46.250" />
<Node x="483207.063" y="3619204.000" z="2.000" Width="10.560" Height="6.188" DirAz="0.000" Speed="2.643" Type="4" ActiveRegions="" Time="2007-02-22T19:41:46.718" />
<Node x="483205.875" y="3619204.750" z="2.000" Width="12.531" Height="6.515" DirAz="0.000" Speed="3.370" Type="4" ActiveRegions="" Time="2007-02-22T19:41:47.187" />
<Node x="483206.156" y="3619205.750" z="2.000" Width="11.668" Height="6.903" DirAz="0.000" Speed="12.402" Type="4" ActiveRegions="" Time="2007-02-22T19:41:47.640" />
<Node x="483207.625" y="3619208.000" z="2.000" Width="10.746" Height="7.108" DirAz="6.212" Speed="19.005" Type="4" ActiveRegions="" Time="2007-02-22T19:41:48.109" />
</Obj>
<Obj ID="192.168.0.14.1.3366995" ActiveID="192.168.0.14.1.3366995">
<Node x="483296.375" y="3619251.250" z="2.000" Width="0.949" Height="1.745" DirAz="0.000" Speed="0.960" Type="4" ActiveRegions="" Time="2007-02-22T19:42:29.468" />
<Node x="483296.000" y="3619251.500" z="2.000" Width="1.231" Height="1.942" DirAz="0.000" Speed="8.939" Type="4" ActiveRegions="" Time="2007-02-22T19:42:29.921" />
<Node x="483295.781" y="3619251.750" z="2.000" Width="1.239" Height="1.926" DirAz="0.000" Speed="1.139" Type="4" ActiveRegions="" Time="2007-02-22T19:42:30.390" />
<Node x="483295.906" y="3619252.000" z="2.000" Width="1.033" Height="1.778" DirAz="5.498" Speed="0.519" Type="4" ActiveRegions="" Time="2007-02-22T19:42:30.843" />
</Obj>
</CamAData>
```

Figure 9: Vistascape Sensor Data Management System Message



```xml
<SMS>
  <SensorMessages Site="MDA" Region="US" DeviceType="MDA" Type="TRACK"
                  MessageTime="2008-12-15T09:11:04.34375-08:00"
                  TrackCategory="SEA" ServerName="10.96.12.68">
    <Info>
      <Id>345030051</Id>
      <SensorStatus>CONNECTED</SensorStatus>
      <OriginalTime>2008-12-15T16:23:07.53125Z</OriginalTime>
      <SeaProperty>
        <Callsign>0</Callsign>
        <MMSI>345030051</MMSI>
        <IMO>7807299</IMO>
        <TrackName>345030051</TrackName>
      </SeaProperty>
      <Parameters Name="defaultInformationSource">SMS</Parameters>
      <Parameters Name="isDistinguishingRawFromAugmentedValues">False</Parameters>
      <Parameters Name="start">12/15/2008 4:23:08 PM</Parameters>
      <Parameters Name="end">12/15/2008 4:23:08 PM</Parameters>
      <Parameters Name="rate">0</Parameters>
      <Parameters Name="eRate">0</Parameters>
      <Parameters Name="PortOfCallName" />
      <Parameters Name="draft">0</Parameters>
      <Parameters Name="signalStrength">0</Parameters>
      <Parameters Name="aisVersionIndication">0</Parameters>
      <Parameters Name="receiverFromBow">0</Parameters>
      <Parameters Name="receiverFromStern">0</Parameters>
      <Parameters Name="receiverFromPortBeam">0</Parameters>
    </Info>
    <Location>
      <Longitude>-117.237438500276</Longitude>
      <Latitude>32.4075810282011</Latitude>
      <Altitude>32.4075810282011</Altitude>
      <Heading>0</Heading>
      <Course>80.9000015258789</Course>
      <Speed>1</Speed>
    </Location>
  </SensorMessages>
</SMS>
```

Figure 10: SMS Sensor Message Format

After message translation is complete, SMS will push each translated track message into a message queue called the *AggregateSensorTrackQueue*. This message queue will contain all tracks originated from the surveillance sensors integrated to SMS. When a client system such as JPSC2 requests sensor track messages from SMS, it calls the *SensorDataWS* Web service in SMS. Upon receiving the track request message from the client system, *SensorDataWS* forwards the request to a .NET component called *SensorRemoteObj*. *SensorRemoteObj* in turn will retrieve sensor tracks from the *AggregateSensorTrackQueue* and return the retrieved tracks to *SensorDataWS*. *SensorDataWS* then will return the received tracks to the requesting client system.

Now, we have examined how data flows from sensors to SMS; let's examine how data flows from an external C2 system such as JPSC2 to sensor systems via SMS.

Let's consider this scenario: a watch officer brings up a camera window for *Camera A* from the JPSC2 interface and clicks on the "*pan left*" button in the camera window. After the button has been pressed, a "*pan left*" command is sent to SMS through its *SensorControlWS Web service* interface. Upon receiving the "*pan left*" command from JPSC2, *SensorControlWS* forwards the command to *SensorRemoteObj*. *SensorRemoteObj* in turn forwards the command to the corresponding software library that communicates with *Camera A*. Upon receiving the "*pan left*" command, the software library translates this generic "*pan left*" command into the proprietary message format that *Camera A* understands and then forwards the command to Camera A through a serial interface (which is the communication interface that C*amera A* uses to communicate with remote systems). *Camera A* then pans to the left.

In the above scenario, we can see that a SMS sensor library not only translate sensor track data to SMS message format, but also translate SMS messages into sensor specific messaging protocols. In other words, SMS sensor libraries can handle two-way translations between SMS and the surveillance sensor systems.

B. INTERFACE TO EXTERNAL C2

In this section, we will present SMS's communications interface to external C2 systems. As Figure 6 shows, all SMS communication with external C2 systems is carried out via Web services. We shall explore two sets of SMS Web service interfaces: 1) *SensorDataWS* Web service interface and 2) *SensorControlWS* Web service interface.

Before diving into the implementation details of the two Web services, we want to make some comments regarding their design approach. When being called, both of the two Web services delegate their actual service logic implementations to a .NET component named *SensorRemoteObj* through .NET Remoting. For example, when the *RegisterData* Web service method in *SensorDataWS* is called by a client system, *SensorDataWS* calls the *RegisterData* method in *SensorRemoteObj* to carry out the actual registration of the client system. *SensorDataWS* itself does not implement any service logic to register the client, all registration programming logic are implemented by

SensorRemoteObj. In this context, Web services are implemented as component wrappers. Its primary role is to introduce an integration layer that consists of wrapper services that enable synchronous communication via SOAP-compliant integration channels.

1. Sensor Data Interface

The *SensorDataWS* Web service interface consists of a set of Web service calls to acquire tracks detected by surveillance sensors integrated to SMS. The interface contains the following Web service method calls: (A detailed interface description (WSDL) of the *SensorDataWS* Web service is listed in the Appendix)

String RegisterData(String clientName,
String format,
String filterType,
String filterString)

The *RegisterData* Web service method registers a client system with SMS. Any client system that wants to receive track messages from SMS needs to register with SMS first. If registration is successful, the method will return the string "SUCCESSFUL", if not, the method will return error messages indicating why failure occurred.

The *clientName* parameter specifies the name of the client. The format parameter specifies the type of messages that the client system is interested in. There are three types of messages in SMS: track, sensor status, and incident. Track messages are target position data reported by the sensors; status messages indicate the status of the sensor ("*on*" or "*off*"); incident messages are special messages that represent critical events such as intrusion of a protected zone that the sensor detected. They usually come into the JPSC2 system as alerts. The *filterType* and *filterString* parameters allow the client system to filter track messages based on sensor specific properties such as device type (AIS, PSRS ground radar, video detection device, etc.), sensor location, etc.

String UnregisterData(String clientName)

The *UnregisterData* Web service method unregisters a client system with SMS. Once the client system unregisters with SMS, it no longer receives data from SMS. If unregistration is successful, the method will return the string “SUCCESSFUL,” if not, the method will return error messages indicating why failure occurred.

The *clientName* parameter is the name that the client used to register with SMS.

String GetMessage(String clientName)

The GetMessage Web service method receives messages from SMS once it has successfully registered with SMS.

The string *clientName* is the name that the client system used to register with SMS.

String GetClientList()

This method returns a list of SMS’s clients.

String GetSensorList()

This method returns a list of sensors that SMS is currently managing.

With the *SensorDataWS* Web service interface defined and explained, let’s now examine in detail how external systems interact with SMS to receive tracks via the *SensorDataWS* Web service interface.

Before client systems can receive tracks from SMS, they need to call the *RegisterData* Web service method to register with SMS. When the *RegisterData* Web service method is called, it delegates the registration process to the *SensorRemoteObj* component in SMS. If registration is successful, the client system will call the *GetMessage* Web service method. This method will in turn call *SensorRemoteObj* to verify whether the client system is indeed registered. If verification has succeeded, *SensorRemoteObj* then creates a message queue for that client system, and populates the queue with track messages from the *AggregateSensorTrackQueue* based on the client system’s message filters defined during the registration process (*filterType* and *filterString* parameters in the *RegisterData* method). As mentioned earlier, the *AggregateSensorTrackQueue* contains all track messages originated from all currently active sensors, and the *RegisterData* Web service method allows the client system to define what type of messages to retrieve from SMS. There is exactly one track message queue for every client system that tries to receive tracks from SMS. This queue is created only when the client system has successfully passed SMS’s authentication process. This queue will be active as long as the client system's registration is valid. When the client system unregisters from SMS, the queue and all the messages in it will be dropped. Another way for the queue to get de-allocated is when the client system stops calling the *GetMessage* method for more than 15 minutes, then the queue will also be dropped by SMS.

2. Sensor Control Interface

The sensor control interface consists of a set of Web service calls to send control commands to the sensors. The interface contains the following Web service method calls: (A detailed interface description (WSDL) of the *SensorControlWS* Web service is listed in the Appendix)

String RegisterControl(String clientName,
String sensorSite)

The *RegisterControl* Web service method registers a client system with SMS. Any client system that wants to send control commands to SMS sensors needs to register with SMS first. If registration is successful, the method will return the string "SUCCESSFUL", if not, the method will return error messages indicating why failure occurred.

The *clientName* parameter specifies the name of the client. The *sensorSite* parameter specifies the name of the sensor that the client system wants to communicate to.

String UnregisterControl(String clientName)

The *UnregisterControl* Web service method unregisters a client system with SMS. Once the client system unregisters with SMS, it no longer can send control commands to the sensor via SMS. If unregistration is successful, the method will return the string "SUCCESSFUL", if not, the method will return error messages indicating why failure occurred.

The *clientName* parameter is the name that the client used to register with SMS.

String SendCommand(String clientName,
String sensorSite,
String command)

The *SendCommand* Web service method is called by the client system to forward sensor control command to the corresponding sensor system via SMS. The command passed from client systems are generic commands. Client systems do not know any sensor specific information such as the sensor’s communication protocol and messaging format. They do not need to be concerned about coding anything sensor specific. They only needs to speak the “language” that SMS speaks. SMS will handle the translation from generic commands to sensor specific commands based on the sensor’s communication interface.

The *clientName* parameter is the name that the client system used to register with SMS. The *sensorSite* parameter specifies the name of the sensor that the client system wants to communicate with. The command parameter specifies the control command that the client system wants to send to the sensor.

String GetSensorInfo()

The GetSensorInfo Web service method will return all current sensor status (up or down) to the client system.

String GetSensorInfoByName(String sensorSite)

The *GetSensorInfoByName* Web service method will return the current sensor status (up or down) of a given sensor to the client system. The *sensorSite* parameter specifies the name of the sensor that the client system wants to get information on.

String GetSensorList()

This method returns a list of sensors that SMS is currently managing.

With the Sensor Control Web service interface defined and explained, let’s examine how external systems interact with SMS to control sensors.

Before client systems can send a control command to sensors via SMS, they need to call the *RegisterControl* Web service method to register with SMS. When the *RegisterControl* method is called, it delegates the registration process to the *SensorRemoteObj* component in SMS. If registration is successful, the client system will call the *SendCommand* method. This method will in turn call *SensorRemoteObj* to verify whether the client system is indeed registered. If verification has succeeded, *SensorRemoteObj* will forward the control command to the appropriate sensor library loaded in SMS. The sensor library in turn will translate the generic command into the sensor's specific message format and send it to the sensor system via its remote communication interface. The communication channel established between the client system and the managed SMS sensor is active as long as the sensor control registration for the client system is valid. When the client system unregisters from SMS by calling the *UnregisterControl* Web service method, the communication path between the client system and the sensor will be destroyed. Another way for the communication path to get dropped is when the client system stops calling the *SendCommand* Web service method for more than 15 minutes, then the client system has to call *RegisterControl* Web service again to reestablish communication with the sensor.

IV. SOA ANALYSIS OF SMS

From the previous chapter, we can see that SMS provides a set of Web service interfaces to communicate with client systems. Does this implementation of Web services automatically make SMS's design service-oriented? In this chapter, we shall carry out the analysis and try to answer that question.

As mentioned earlier in the thesis, SOA is a concept. When we evaluate a system's architecture to decide whether it is based on SOA, what we are really determining is whether the system's architectural design follows the SOA design principles. The evaluation of a system's service orientation is not a clear cut process. Most systems have some design features that follow the SOA principles and some design features that do not. Thus the best way to evaluate a system's architecture is not to determine whether it is based on SOA but to determine how well its architecture follows SOA design principles. In other words, what we are really evaluating is the system's degree of service-orientation.

We believe that software design and software architecture evaluation are both heuristic processes. Both depend on experience-based techniques, educated guesses, and intuitive judgments. Software architecture evaluation based on a quantitative approach still has to depend on subjective assessment. Thus a qualitative approach to evaluate SMS's degree of SOA will be adopted in this thesis.

There is no official set of service-orientation principles. There are, however, a common set of principles most associated with service-orientation. We have defined those principles in Chapter II Section A. They will form the basis upon which the SMS's degree of service-orientation will be evaluated.

Before we go into each service-orientation principle to evaluate SMS, let's examine some of the design characteristics of SMS's Web service interface.

As described in earlier sections, and also as Figure 6 shows, SMS has two sets of Web service interfaces: *SensorDataWS* and *SensorControlWS*. When being called, both of those two Web services delegate their service implementations to a .NET component named *SensorRemoteObj* through .NET Remoting. For example, when the *RegisterData* method in *SensorDataWS* is called by a client system, *SensorDataWS* calls the *RegisterData* method in *SensorRemoteObj* to carry out the actual registration of the client system. *SensorDataWS* itself does not implement any service logic to register the client, all registration programming logic are implemented by *SensorRemoteObj*. In this context, Web services are implemented as component wrappers. Its primary role is to introduce an integration layer that consists of wrapper services that enable synchronous communication via SOAP-compliant integration channels.

These integration channels are primarily utilized in integration architectures to facilitate communication with other applications. They can also be used to enable communication with other (more service-oriented) solutions and to take advantage of some of the features offered by third-party utility Web services. It is important to clarify that a distributed architecture that incorporates Web services in this manner does not qualify as a true SOA. It is simply a distributed architecture that uses Web services [2].

Web services within SOA are subject to specific design requirements, such as those service-orientation principles specified in Chapter II. These and other characteristics support the pursuit of consistent loose coupling. Once achieved, a single service is never limited to point-to-point communication; it can accommodate any number of current and future requestors.

Now, let's examine SMS's degree of service-orientation based on SOA design principles.

1. SMS Architecture Shares a Formal Contract

Communication between SMS and client systems is carried out via Web services. SMS provides Web service interfaces for client systems to either receive sensor track data from or send control commands to sensor systems. The WSDL files for SMS's Web service interfaces are the formal contracts that bind the service requesters (such as JPSC2) and the service provider (SMS).

Also, all sensor software libraries in SMS implement the same interface. This interface is a formal contract between the sensor libraries and SMS infrastructure components that invoke individual sensor libraries to get sensor track feeds. In other words, you can view SMS infrastructure components as service requesters and the sensor libraries as service providers. They are bind to one another through the sensor library interface.

2. SMS Architecture is Designed to Abstract Underlying Logic

As described in the last section, when a JPSC2 operator brings up a camera control window, and he clicks on the "*pan left*" button, a "*pan left*" command along with the camera's model are sent to SMS via its *SensorControlWS* Web service interface. Upon receiving this command, SMS calls the software library specifically written for that camera model. Since the camera only understands its vendor specific protocol, the SMS software library for that camera model translates the generic command "*pan left*" originated from JPSC2 to the camera vendor's own protocol "*pan left*" command and send it to the camera. The camera then pans to the left.

From the above example, we can see that JPSC2 does not have to know all of the sensors' unique communication protocols. It only needs to speak the "language" that SMS speaks. SMS abstracts all of the programming logic to communicate with specific sensors, so that client systems such as JPSC2 do not. This design allows the decoupling of the user interface (JPSC2) and the communication backend. When sensors need to be added, modified, and deleted, only individual sensor libraries will be modified, the user interface portion of the system are left unchanged.

The same design pattern also applies to client systems that receive sensor tracks from SMS. Each sensor system has its own communication protocol and messaging format. SMS communicates with sensor systems using the sensors' unique communication interfaces, translates sensor specific message formats to a generic sensor message format, and make sensor tracks collected from the various sensors accessible through the *SensorDataWS* Web service interface. Again, client systems only need to speak the language that SMS speaks. SMS abstracts the programming logic to handle unique proprietary sensor specific protocols so that client systems do not. When sensors need to be added, modified, and deleted, only individual sensor libraries will be modified. To external systems, those operations are transparent.

3. SMS Architecture is Deficient in Building Modular Services

SOA represents a distinct approach for separating concerns. What this means is that logic required to solve a large problem can better be constructed, carried out, and managed if it is decomposed into a collection of smaller, related pieces. Each of these pieces addresses a concern or a specific part of the problem [2].

As argued in the earlier section, the Web service interface for SMS is nothing more than a set of component wrappers. All SMS Web services delegate their service implementations to the *SensorRemoteObj* .NET component. This single component's functionalities can be logically partitioned into four separate and independent operations:

- Register Client Systems
- Authenticate Client Systems
- Retrieve sensor track messages from the *AggregateSensorTrackQueue*
- Forwarding sensor control commands from client systems to sensor libraries

All of the above operations have distinct separation of concerns, thus each one of the four major operations should be implemented as autonomous and independent services.

4. SMS Architecture is Deficient in Building Autonomous Services

A service can be viewed as an independent software program that realizes a set of functionalities. As each service might have an architecture that is different from the others, it needs to be designed individually.

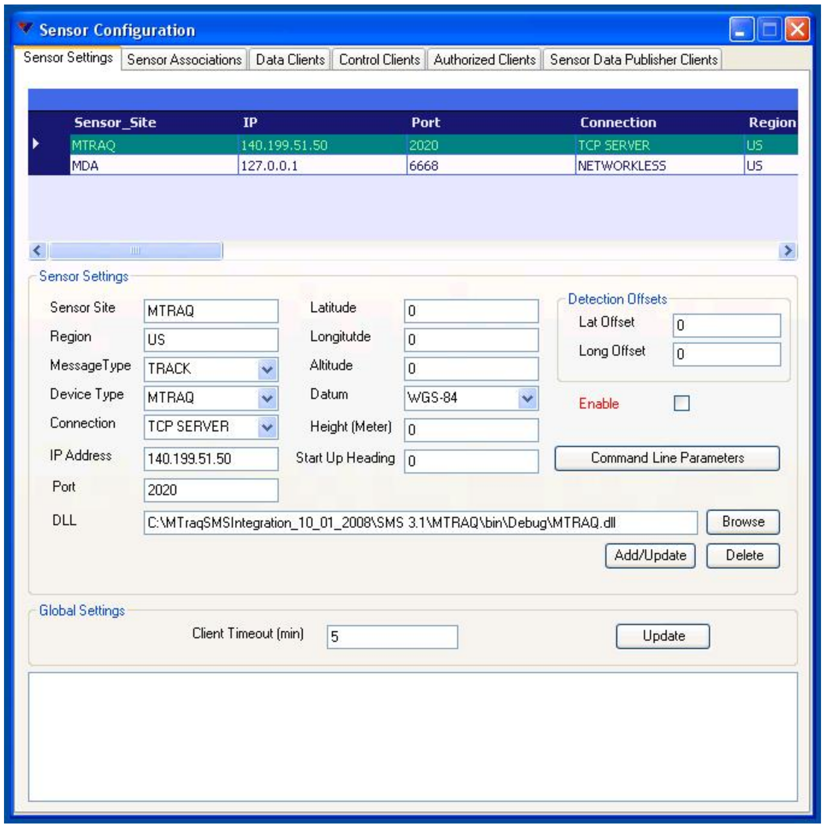
a. SMS Web Services are not Autonomous

Again, Web services under current SMS architecture are nothing more than component wrappers. They are not real services that have complete control over the logic they encapsulate. They depend on *SensorRemoteObj* to implement their service logics. *SensorRemoteObj* is a completely independent software entity that was not specifically written for the SMS Web services. SMS Web services were actually developed to expose SMS's interface via Web services. Thus SMS Web services are not autonomous.

b. SMS Sensor Libraries are not Autonomous

Sensor libraries in SMS depend heavily on other SMS services to be loaded, run, and de-allocated. Let's examine in detail how SMS sensor libraries are governed by SMS's run-time infrastructure.

SMS provides a software tool to allow SMS system administrators to load sensor libraries into SMS runtime space. Figure 11 shows a snapshot of the user interface for the tool.



Sensor_Site	IP	Port	Connection	Region
MTRAQ	140.199.51.50	2020	TCP SERVER	US
MDA	127.0.0.1	6668	NETWORKLESS	US

Sensor Settings

Sensor Site	MTRAQ	Latitude	0
Region	US	Longitutde	0
MessageType	TRACK	Altitude	0
Device Type	MTRAQ	Datum	WGS-84
Connection	TCP SERVER	Height (Meter)	0
IP Address	140.199.51.50	Start Up Heading	0
Port	2020		
DLL	C:\MTraqSMSIntegration_10_01_2008\SMS 3.1\MTRAQ\bin\Debug\MTRAQ.dll		

Detection Offsets

Lat Offset	0
Long Offset	0

Enable ☐

Global Settings

Client Timeout (min) 5

Figure 11: SMS Admin Tool User Interface

When a SMS system administrator configures SMS to communicate with a particular sensor, the administrator enters, at a minimum, the following sensor settings to load a given sensor library:

SensorSite: Name of the sensor

Region: Geographical region where the sensor belongs

MessageType: Message type that the sensor reports (track, status, or incident)

DeviceType: Name of Sensor Library

Connection: Type of communication protocol of the sensor

IP Address and Port: IP address and port that SMS uses to communicate with the sensor track system

DLL: the actual *.dll(dynamic link library)* file of the sensor library

As explained in the last section, SMS is implemented as a Windows service, and each sensor library is implemented as a Dynamic Link Library (*dll*). SMS has some infrastructure components that provide the run-time governess infrastructure for individual sensor libraries. When the SMS system administrator clicks on the "*Add/Update*" button in the SMS Admin Tool after entering sensor settings, some SMS infrastructure component spawns a thread that establishes the network connection with the remote sensor system as defined in the sensor settings and then loads the sensor library *.dll* file into SMS's runtime space via .NET Reflection. After network connection to the sensor has been established and sensor library has been loaded successfully, the same SMS infrastructure component will call the *GetSensorMsg* method in the loaded sensor library to process the sensor track messages and push the processed messages into the *AggregateSensorTrackQueue*.

We can see that it is the SMS infrastructure components that establish network communication to the sensor, not the SMS sensor libraries. The sensor communication protocol is part of the sensor system's interface; it is an inherent property and a specific attribute of the sensor. In other words, establishing communication with the sensor should be individual sensor library's concern, not other SMS infrastructure components' concern. By including sensor specific attributes in the sensor library, the sensor libraries become self-describing, more autonomous and independent from SMS infrastructure components.

Also, under current design, SMS sensor libraries do not push processed sensor track messages into the *AggregateSensorTrackQueue*. This service is provided, again, by SMS's infrastructure components. If a change is made to the SMS infrastructure components and the change cause the infrastructure components to stop functioning, then no sensor libraries can push messages into the *AggregateSensorTrackQueue*. From a service autonomy point of view, pushing messages into the track queue should be a function that is provided by individual sensor libraries.

Overall, the current approach makes SMS sensor libraries less autonomous since they depend on other SMS services to provide essential functionalities and runtime environment.

5. SMS Design has Services that are Stateful

Under current SMS Web service interface design, when a client system sends a request to receive sensor track messages from SMS, it needs to register with SMS first by calling the *RegisterData* Web service method. The *RegisterData* method defines the message format and message filter for the subscribed sensor track messages. Once the *RegisterData* Web service method has been successfully called, SMS creates a session for the requesting client system. The session retains the message filter and message format information specified by the client system in the *RegisterData* Web service call. Once the client system successfully establishes a session with SMS, it's ready to call the *GetMessage* Web service method to receive sensor track messages. When the *GetMessage* method is called by the client system, SMS creates and maintains a sensor message queue for the requesting client system, and populates the message queue with messages defined by the message filter and message format specified in the *RegisterData* Web service call.

As described above, SMS has to maintain session information between Web service calls. This approach makes SMS service design stateful. The successful execution of *GetMessage* depends on the session information (message format and filter) created by the *RegisterData* method. Services should be independent, self-contained requests, which do not require information or state from one request to another when implemented. Services should not be dependent on the context or state of other services. When dependencies are required, they are best defined in terms of common business processes, functions, and data models, not implementation artifacts (like a session key). Sometimes service requesters require persistent state between service invocations, but this should be separate from the service provider.

6. SMS Architecture is Deficient in Building Loosely Coupled Services

a. Sensor Systems and SMS

As discussed earlier, SMS sensor libraries are software components (Microsoft Dynamic Link Libraries) implemented using the .NET framework. When integrating a new sensor system into SMS, a developer has to develop a sensor software library for that sensor system based on the communications interface and the messaging format for the sensor system.

This level of dependence is a form of tight-coupling. If the communications interface of the sensor system changes, then the implementation of the corresponding SMS sensor library will have to change accordingly. The main problem is that the communication between a sensor system and SMS is based on that sensor's communication interface instead of an open, standardized interface such as Web services.

b. Stateful Transaction

As discussed earlier, SMS has Web service methods that are stateful. The *RegisterData* method creates a session for the calling client system, and the *GetMessage* method depends on the session variables created by the *RegisterData* method for its successful execution. This approach makes the *GetMessage* method tightly coupled with the *RegisterData* method.

c. Sensor Libraries and SMS Infrastructure Components

Since sensor libraries depend on SMS infrastructure components to provide services such as runtime governess (load and run), establishing network connection to sensor systems, and pushing processed sensor messages to the *AggregateSensorTrackQueue*, sensor libraries and SMS infrastructure components are tightly coupled.

7. SMS Architecture is Deficient in Building Reusable Services

As argued earlier, since the *SensorRemoteObj* module is not decomposed properly based on separation of concerns, its reusability is very limited.

For example, authenticating client systems is an inherent part of the programming logic that *SensorRemoteObj* implements; there does not exist a separate module that handles client authentication. *SensorRemoteObj* was not specifically designed to carry out client authentication. It performs many other tasks such as receiving sensor track messages and providing the infrastructure for sensor libraries to load and run. Reusable components should have very high degree of cohesion; it should do one thing and one thing only. *SensorRemoteObj* clearly violates that design principle.

SMS should not embed authentication logic in its application code. This approach does not scale well. If JPSC2 needs to access other services in SMS that require authentication, then these services will have to replicate the authentication code currently implemented in *SensorRemoteObj*.

Client authentication and sensor management are two separate logical entities that have their own distinct services to fulfill. Sensor management provides the services to manage sensors, and user authentication provides the service to authenticate client systems. By making *SensorRemoteObj* implementing client authentication logic instead of delegating it to a highly cohesive authentication service, *SensorRemoteObj* becomes tightly coupled with the client authentication process which lowers the potential for reusability.

8. SMS Architecture is Deficient in Building Composable Services

Again, service composability is not possible if services are not designed to be loosely coupled, reusable, and modular based on separation of concerns. As argued earlier, the *SensorRemoteObj* module within SMS is deficient in loose-coupling, reusability, and modularity, thus SMS is deficient in service composability as well.
For example, the *SensorRemoteObj* module performs four major tasks:

- Registering Clients
- Authentication Clients
- Retrieving Sensor Track Messages from *AggregateSensorTrackQueue*
- Sending Sensor Commands to Sensors via SMS sensor libraries

All of the above tasks are lumped together into one single service. Since each task in the above list is an independent and separate logical entity, *SensorRemoteObj* should be decomposed into at least those four separate services. Then those services can be composed to fulfill different business requirements. For example, the service that retrieves sensor track messages and the client authentication service can be composed to form a Sensor Data Service to service client requests on sensor track messages.

9. SMS Design does not Support Service Discovery

Under current SMS architecture, there is no mechanism to advertise and discover services. A service registry or directory for storing and managing service descriptions does not exist.

V. ALTERNATIVE DESIGN BASED ON SOA

In this chapter, we will present some alternative design approaches that will increase the degree of service-orientation for SMS.

Figure 12 shows the high level design of our proposed architecture. Under this new architecture, the *SensorRemoteObj* component in SMS is decomposed into more granular services (*Sensor Data Web Service, Sensor Control Web Service,* and *Sensor Data Publisher Web Service*) to increase the system's overall modularity; a *User Authentication Service* is added to make the overall system design based on a better scheme of separation of concerns; a Database Management System is also added to reduce the burden of managing stateful information; new web services at each sensor system (*Sensor Control Receiver Web Service* and *Sensor Data Provider Web Service*) and SMS (*Sensor Data Publisher Web Service*) are established to make system integration more loosely coupled; and finally a UDDI registry is added to make all the system's available Web services discoverable.

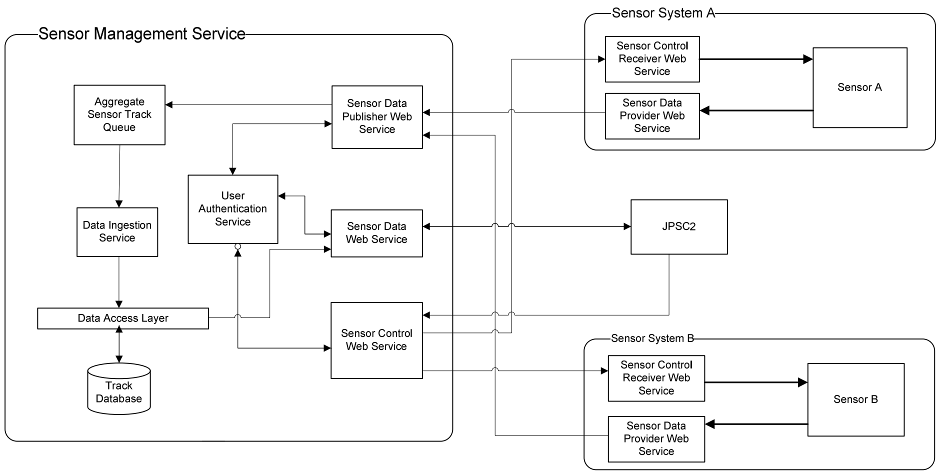

Sensor Management Service
Aggregate Sensor Track Queue
Sensor Data Publisher Web Service
User Authentication Service
Data Ingestion Service
Sensor Data Web Service
Data Access Layer
Sensor Control Web Service
Track Database
Sensor System A
Sensor Control Receiver Web Service
Sensor Data Provider Web Service
Sensor A
JPSC2
Sensor System B
Sensor Control Receiver Web Service
Sensor Data Provider Web Service
Sensor B

Figure 12: Proposed SMS Architecture

The rest of this chapter will explain in detail how the new designs improve the existing system's service-orientation.

1. Implementing SMS Web Services as Autonomous and Independent Services

As elaborated in the last chapter, Web services are implemented as component wrappers in SMS. All SMS Web services delegate their service implementation logic to a .NET component named *SensorRemoteObj*. This approach makes services less autonomous, less modular, and less reusable. It also makes service logics tightly coupled.

To improve the design, we first need to build SMS Web services as autonomous and independent services, not as service wrappers around a single component. This means we need to remove *SensorRemoteObj* from SMS and implement the actual service logics in the Web services themselves.

By doing this, not only we are implementing SMS Web services as real services, but we are also decomposing service logics that used to be aggregated in one single component into a set of services based on separation of concerns. *SensorDataWS* and *SensorControlWS* are modules that are designed to perform distinct operations based on separation of concerns (registration, receiving message, send command, etc.). By removing *SensorRemoteObj* from SMS and decompose its service logics to form autonomous services, the entire system becomes more modular.

2. Building Sensor Libraries as Autonomous Services

As mentioned in the last chapter, sensor libraries in SMS are .NET components (Dynamic Link Library files) that depend on SMS's infrastructure services to load and execute. Their design follows the traditional component architecture. They are not written as autonomous and independent services loosely coupled from SMS and they cannot be composed to form larger services to provide a variety of sensor track messages.

In addition, under the current architecture, when SMS integrates a new sensor, the corresponding SMS sensor library has to know the sensor system's specific communication protocol, messaging format, and remote interface in order to establish communication with the sensor system and convert sensor specific message format to SMS message format. This approach makes SMS tightly coupled with the sensor systems. Every time a sensor system changes its communications interface, the corresponding SMS sensor library has to be changed. In other words, the sensor system's communication interface dictates the implementation of SMS and its corresponding sensor libraries.

We propose an alternative system architecture to integrate sensor systems with SMS. First, we develop a new Web service interface for SMS. We call this Web service *SensorDataPublisherWS*. The *SensorDataPublisherWS* service provides an interface for other client sensor systems to publish sensor data to SMS. Below is a service description on *SensorDataPublisherWS*: (The Appendix shows a detailed description (WSDL) of the Web service methods for the Sensor Data Publisher Web Service Interface.)

String PublishSensorData(String clientName,
String sensorMessages)

The *PublishSensorData* Web service method is called by the client sensor system to push sensor data into SMS.

The *clientName* parameter is the name of the client sensor system. Before a client sensor system can publish its tracks to SMS, its name has to be stored in SMS's client sensor data store. The *sensorMessages* parameter contains an array of sensor messages in SMS message format (Figure 10) that will be published to SMS.

Next, we develop an integration service for each sensor system that wants to provide track data to SMS. We shall call this service the *SensorDataProviderWS* service and it performs the following tasks:

1. Establish communication with the sensor system based on the sensor system's communication interface.
2. Convert the sensor system's unique proprietary message format into SMS track message format.
3. Call the *PublishSensorData* method in *SensorDataPublisherWS* to publish the converted sensor track messages to SMS.

The *SensorDataProviderWS* service defined for each sensor system together with the *SensorDataPublisherWS* defined in SMS, implement the message processing functionalities. In other words, the *SensorDataProviderWS* shifts the responsibility of receiving and processing sensor track data from SMS to individual sensor systems.

Finally, we develop a Web service interface for each sensor system that receives sensor control commands from SMS. We shall call this service the *SensorControlReceiverWS* service:

1. When SMS receives a control command from JPSC2 via its *SensorControlWS* Web service, it forwards that command to the appropriate sensor system by calling that sensor system's *SensorControlReceiverWS* service.
2. Upon receiving the sensor control command, the *SensorControlReceiverWS* service converts the command message from the generic SMS message format to the sensor system's unique proprietary message format.
3. After the command message has been converted, the *SensorControlReceiverWS* service establishes communication with the sensor system and sends the command to the sensor system via the sensor's communications interface.

The *SensorControlReceiverWS* basically shifts the responsibility of processing sensor control commands from SMS to individual sensor systems.

So the question is why are we doing this? As we have discussed in earlier sections, one of the fundamental characteristics of SOA-based service "modules" is that they are constructed with loosely-coupled interfaces to allow for business process flexibility and use in multiple business processes. When properly designed, loosely coupled services support a composition model, allowing individual services to participate in aggregate assemblies. This introduces continual opportunities for reuse and extensibility.

Under the proposed architecture, the *SensorDataProviderWS* service and the *SensorControlReceiverWS* services make each sensor system an autonomous, independent, and composable service. Since each sensor system becomes a service, a variety of sensor systems can be composed to provide a variety of sensor track data.

Secondly, the *SensorDataPublisherWS* Web service provides an interface for any third party sensor systems to integrate with SMS via an open, standardized interface—irrespective of the technology used to implement the underlying logic. The standardized interface supports the open communications framework that sits at the core of SOA. The use of Web services establishes a framework under which building loosely coupled software services is greatly simplified.

Thirdly, the proposed architecture makes sensor system's communication protocol loosely coupled from SMS's internal implementation. The change of individual sensor system's implementation and communication interface would not affect the implementation of SMS. SMS does not care what the sensor system's communication interface is or what message format it uses because it is the responsibility of the sensor system's Web services (*SensorDataProviderWS* and *SensorControlReceiverWS)* to publish sensor tracks and process sensor control commands using SMS's communication protocol (Web services) and messaging format.

3. Implementing Client Authentication as a Separate Service

Under current SMS architecture, client authentication code is embedded inside of SMS's application code. User authentication and sensor management are two separate logical entities that have their own distinct services to fulfill. Sensor management provides the services to manage sensors, and user authentication provides the service to authenticate client systems. The client authentication mechanism should be implemented as an autonomous and independent software module separated from sensor management application logics. This approach would improve the modularity, autonomy, and reusability of both modules.

There are many ways to implement the authentication service; we will discuss two possible methods:

1) Create a *UserAuthentication* Web service that has the following method:

String UserAuthenticate(String username)

The username parameter provided by the method will be checked against a database that stores all SMS client authentication credentials. The method will return SUCCESSFUL if authentication is passed, otherwise it will return FAILED. All authentication code is contained within this service. Under this scheme, there is no mixing of sensor management logic and user authentication logic. Service requestors of this service simply compose this service to authenticate their clients.

2) Another approach, which is a better approach, is to use a Web service container that provides not only user authentication but also wire-level security. The Internet Information Service (IIS), Microsoft's Web server, can provide both.

This obvious benefit of this approach is that authentication mechanism is implemented by the Web server container, rather than the application. It is the IIS rather than the application that becomes the security provider.

This approach improves modularity since the service can focus on application logic instead of implementing programming logic on security. It also leverages the reusability of the Web server container. A Web server such as IIS can host a variety of applications regardless of their application domains. The security features it provides can be reused by many services and applications.

4. Removing Registration from Web Service Interface

The SMS Web service interface can be changed to minimize state information management. As explained in the last chapter, a client system calls the *RegisterData* method to define what type of messages it wants to receive. When this method is called, state information such as message type and message filter is maintained as session variables by SMS. Those session variables are used by the *GetMessage* method to retrieve messages.

To eliminate state information, The *RegisterData* method should be eliminated. Instead, the *GetMessage* Web service method provides the interface to allow client systems to define what type of messages to subscribe. This approach would eliminate the need to maintain state information in SMS. Below contrasts the current *GetMessage* signature to the proposed signature:

Current signature of *RegisterData() and GetMessage()*:

```
String RegisterData(String clientName,
                    String format,
                    String filterType,
                    String filterString)
String GetMessage(String clientName)
```

Proposed signature of *GetMessage*:

String GetMessage(String clientName,
String filterType,
String filterString)

When the client system calls the new *GetMessage* method, *GetMessage* just retrieves the messages from the data store based on the definition of the message filter defined in the call. If the client system chooses to subscribe to a different set of messages, all it needs to do is to define a different set of message filter parameters to reflect the change. No re-registration is required.

5. Adding DBMS to Minimize State Information

As explained in Chapter III, upon calling the *GetMessage* method, SMS creates and maintains a sensor message queue for the requesting client system, and populates the message queue with messages defined by the message filter and message format specified by the client system. The client message queue does not get created unless *GetMessage* is called by a requesting client, and there is a one-to-one relationship between the number of client message queues and the number of clients. If SMS has *n* clients, there will be potentially *n* client message queues to manage. Obviously, this approach does not scale very efficiently. Client message queue created for each client is considered state information that should be eliminated to promote service-orientation.

The proposed solution is to store sensor track messages in a DBMS instead of storing them in message queues. We can build a *Data Ingestion Service* whose function is to do the following:

1. retrieves messages from SMS's *AggregateTrackMessageQueue* where all sensor track messages are stored in SMS message format
2. correlates the retrieved sensor track messages
3. stores the correlated sensor track messages in a central DBMS

The *SensorDataWS* Web service interface remains the same. To client systems, the internal change to SMS is transparent. When a client system calls the *GetMessage* method in *SensorDataWS*, SMS goes to the DBMS to retrieve the requested messages using standard SQL. Thus, instead of managing *n* message queues for *n* request client systems, we now have a central repository that stores all tracks that are accessible through a standard interface. This approach significantly improves system scalability, decouples client system call to SMS internal implementation, eliminates state information to promote loose coupling.

6. Implementing UDDI to Make Web Service Discoverable

As we have established in earlier sections, the sole requirement for one service to contact another is access to the other service’s description. Under the proposed architecture for SMS, as the amount of Web services increase within and outside of SMS/JPSC2 system boundary, mechanisms for advertising and discovering service descriptions may become necessary. A central directory and registry such as UDDI should be used to keep track of the many service descriptions that become available.

A UDDI registry can be used to:

- Locate the latest versions of known service descriptions
- Discover new Web services that meet certain criteria

For example, when a new *SensorControlReceiverWS* Web service with a method to accept sensor control commands for a sensor system is developed, its service location and description are advertised in a public UDDI registry. Then, SMS can access the registry to locate the new Web service and calls it to forward sensor control command triggered from JPSC2 to the sensor system.

Another example is that SMS advertises its *SensorDataWS* and *SensorDataPublisherWS* Web services in a UDDI registry. Then any authorized third party sensor systems can locate the *SensorDataPublisherWS* in the registry and publish its tracks to SMS. SMS clients such as JPSC2 can locate *SensorDataWS* in the UDDI registry, and call it to receive SMS managed tracks and display them on the C2 map console.

The implementation of the UDDI registry makes reusable components more readily available to service requestors. The whole process of locating and binding to reusable services becomes very dynamic.

VI. CONCLUSION AND FUTURE WORK

A. CONCLUSION

In this thesis, we presented a case study of a Sensor Management System to investigate the degree of service-orientation of a SOA-based software systems and ways to increase the degree of service-orientation of a software architecture. Through the detailed case study, we tentatively answered the following questions:

1. Does a system's use of Web services make its architecture service-oriented?
2. What determines whether a system is designed based on SOA?
3. What criteria can be used to evaluate a system's degree of service-orientation?

The results of this study conclude that the use of Web services alone by a software application does not automatically make it service-oriented. What makes an application services-oriented depends on whether it is designed and implemented based on the fundamental design principles of service-orientation. The nine fundamental design principles of service-orientation specified in Chapter II: modularity, abstraction, loose coupling, autonomy, sharing of a contract, composability, statelessness, reusability, and discoverability can be used as design criteria to evaluate a software system's degree of service-orientation.

B. FUTURE WORK

1. Web Service Performance

The alternative architecture we proposed in this thesis depends solely on the use of Web services to integrate JPSC2, SMS, and the various sensor systems. Because Web services introduce layers of data processing, it is subject to the associated performance overhead imposed by these layers. For example, Web services security measures, such as encryption and digital signing, add new layers of processing to both the senders and recipients of messages.

Thus, it is critical to understand the performance requirements of the system and the performance limitations of the system infrastructure to build a successful solution.

The following tests and analysis should be conducted to evaluate how the use of Web services impact data processing performance on SMS/JPSC2:

- Testing the message processing capabilities of the system environments prior to implementing Web services
- Stress-testing the vendor supplied processors (for XML, XSLT, SOAP, etc.) intended for use

Normally, data processed by a typical Command and Control (C2) system are near real-time. Surveillance systems such as JPSC2 may have higher real time requirements since it deals with live data (live video feeds and detection) and the response time required by the operators is much faster. If the site where the system is deployed has a limited network bandwidth, then the use of web services may not meet the performance requirements, and we may need to sacrifice the degree of service-orientation for better performance by going with a more component based approach such as .NET Remoting [25].

However, there are some achievements in recent years to increase the data processing speed of Web services. For example, intelligent XML parser technology such as the XML-binary Optimized Packaging (XOP) [26] and SOAP Message Transmission Optimization Mechanism (MTOM) [27] and the advent of XML appliances such as IBM DataPower® greatly enhance Web services data processing performance [28]. Web services caching support in some application servers also improve performance significantly [29]. More studies need to be conducted to explore alternative processors, accelerators, or other types of supporting technology to improve data processing performance.

2. Web Service Security

Since Web services expose the system's external interface on the Wide Area Network (WAN), network security becomes an important issue and needs to be addressed accordingly.

a. Beware of Remote Third-party Services

When a remote third party sensor system on a WAN is being integrated to SMS, it calls the *SensorDataPublisherWS* service in SMS to publish its data; WS-Security should be implemented and incorporated into SMS to ensure that third party systems do not compromise the security of the overall system. One way to mitigate this risk is to test the third-party system with a prototype that simulates the anticipated interaction scenarios before going live.

b. Define an Appropriate System for Single Sign-on

Under our proposed architecture, when JPSC2 issues a sensor control command, it calls the *SensorControlWS* service in SMS, and SMS in turn calls the *SensorControlReceiverWS* service at the corresponding sensor system to execute the command. A security model should be designed with single sign-on in mind to establish an efficient integration model. Security credentials transmitted by the *SensorControlWS* service should be mapped to the *SensorControlReceiverWS* service so that the authentication process is streamlined and administration is relatively centralized.

c. Consider the Development of Security Policies

Since we are dealing with diverse systems that have clearly defined Web services interfaces, it might be a good idea to implement Extensible Access Control Markup Language (XACML) [30] or WS-Policy to provide a means of defining policies that determine what the service requestor can and cannot do with the requested service provider operation. A single policy can apply to a variety of applications and services.

For example, we can design a policy to require all service requestors to SMS Web services to digitally sign and encrypt their messages.

One challenge when using policies is the enforcement of policy rules. We need to ensure that a given Web service is actually checking a policy prior to allowing a service requestor access to a resource. One approach is to centralize security into a separate services layer.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: SERVICE DESCRIPTIONS (WSDL)

A. SENSORDATAWS

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions                    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tns="http://Spawar.navy.mil/Code2644/WebServices/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
targetNamespace="http://Spawar.navy.mil/Code2644/WebServices/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <s:schema                                          elementFormDefault="qualified"
targetNamespace="http://Spawar.navy.mil/Code2644/WebServices/">
      <s:element name="RegisterData">
        <s:complexType>
          <s:sequence>
            <s:element  minOccurs="0"  maxOccurs="1"  name="clientName"  type="s:string"
/>
            <s:element minOccurs="0" maxOccurs="1" name="format" type="s:string" />
            <s:element minOccurs="0" maxOccurs="1" name="filterType" type="s:string" />
            <s:element  minOccurs="0"  maxOccurs="1"  name="filterString"  type="s:string"
/>
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="RegisterDataResponse">
        <s:complexType>
          <s:sequence>
            <s:element     minOccurs="0"     maxOccurs="1"     name="RegisterDataResult"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="UnregisterData">
        <s:complexType>
          <s:sequence>
            <s:element  minOccurs="0"  maxOccurs="1"  name="clientName"  type="s:string"
/>
```

```
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="UnregisterDataResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="UnregisterDataResult"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetSensorList">
        <s:complexType />
      </s:element>
      <s:element name="GetSensorListResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetSensorListResult"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetClientList">
        <s:complexType />
      </s:element>
      <s:element name="GetClientListResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetClientListResult"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetMessage">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="client" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetMessageResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetMessageResult"
type="s:string" />
```

```
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
</wsdl:types>
<wsdl:message name="RegisterDataSoapIn">
  <wsdl:part name="parameters" element="tns:RegisterData" />
</wsdl:message>
<wsdl:message name="RegisterDataSoapOut">
  <wsdl:part name="parameters" element="tns:RegisterDataResponse" />
</wsdl:message>
<wsdl:message name="UnregisterDataSoapIn">
  <wsdl:part name="parameters" element="tns:UnregisterData" />
</wsdl:message>
<wsdl:message name="UnregisterDataSoapOut">
  <wsdl:part name="parameters" element="tns:UnregisterDataResponse" />
</wsdl:message>
<wsdl:message name="GetSensorListSoapIn">
  <wsdl:part name="parameters" element="tns:GetSensorList" />
</wsdl:message>
<wsdl:message name="GetSensorListSoapOut">
  <wsdl:part name="parameters" element="tns:GetSensorListResponse" />
</wsdl:message>
<wsdl:message name="GetClientListSoapIn">
  <wsdl:part name="parameters" element="tns:GetClientList" />
</wsdl:message>
<wsdl:message name="GetClientListSoapOut">
  <wsdl:part name="parameters" element="tns:GetClientListResponse" />
</wsdl:message>
<wsdl:message name="GetMessageSoapIn">
  <wsdl:part name="parameters" element="tns:GetMessage" />
</wsdl:message>
<wsdl:message name="GetMessageSoapOut">
  <wsdl:part name="parameters" element="tns:GetMessageResponse" />
</wsdl:message>
<wsdl:portType name="SensorDataWSSoap">
  <wsdl:operation name="RegisterData">
    <wsdl:input message="tns:RegisterDataSoapIn" />
    <wsdl:output message="tns:RegisterDataSoapOut" />
  </wsdl:operation>
  <wsdl:operation name="UnregisterData">
    <wsdl:input message="tns:UnregisterDataSoapIn" />
    <wsdl:output message="tns:UnregisterDataSoapOut" />
  </wsdl:operation>
  <wsdl:operation name="GetSensorList">
```

```
      <wsdl:input message="tns:GetSensorListSoapIn" />
      <wsdl:output message="tns:GetSensorListSoapOut" />
    </wsdl:operation>
    <wsdl:operation name="GetClientList">
      <wsdl:input message="tns:GetClientListSoapIn" />
      <wsdl:output message="tns:GetClientListSoapOut" />
    </wsdl:operation>
    <wsdl:operation name="GetMessage">
      <wsdl:input message="tns:GetMessageSoapIn" />
      <wsdl:output message="tns:GetMessageSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="SensorDataWSSoap" type="tns:SensorDataWSSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="RegisterData">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/RegisterData"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="UnregisterData">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/UnregisterData"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetSensorList">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetSensorList"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetClientList">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetClientList"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetMessage">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetMessage"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:binding name="SensorDataWSSoap12" type="tns:SensorDataWSSoap">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="RegisterData">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/RegisterData"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="UnregisterData">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/UnregisterData"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetSensorList">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetSensorList"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetClientList">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetClientList"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetMessage">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetMessage"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="SensorDataWS">
    <wsdl:port name="SensorDataWSSoap" binding="tns:SensorDataWSSoap">
      <soap:address
location="http://localhost/SPAWARWebServices/SensorDataWS/SensorDataWS.asmx"
/>
    </wsdl:port>
    <wsdl:port name="SensorDataWSSoap12" binding="tns:SensorDataWSSoap12">
```

```
      <soap12:address
location="http://localhost/SPAWARWebServices/SensorDataWS/SensorDataWS.asmx"
/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

B. SENSORCONTROLWS

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions                    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tns="http://Spawar.navy.mil/Code2644/WebServices/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
targetNamespace="http://Spawar.navy.mil/Code2644/WebServices/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <s:schema                                        elementFormDefault="qualified"
targetNamespace="http://Spawar.navy.mil/Code2644/WebServices/">
      <s:element name="RegisterControl">
        <s:complexType>
          <s:sequence>
            <s:element  minOccurs="0"  maxOccurs="1"  name="clientName"  type="s:string"
/>
            <s:element minOccurs="0" maxOccurs="1" name="sensorSite" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="RegisterControlResponse">
        <s:complexType>
          <s:sequence>
            <s:element    minOccurs="0"    maxOccurs="1"    name="RegisterControlResult"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="UnregisterControl">
        <s:complexType>
          <s:sequence>
            <s:element  minOccurs="0"  maxOccurs="1"  name="clientName"  type="s:string"
/>
```

```
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="UnregisterControlResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="UnregisterControlResult"
type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="GetSensorInfo">
      <s:complexType />
    </s:element>
    <s:element name="GetSensorInfoResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="GetSensorInfoResult"
type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="GetSensorInfoByName">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="sensorSite" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="GetSensorInfoByNameResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1"
name="GetSensorInfoByNameResult" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="GetSensorList">
      <s:complexType />
    </s:element>
    <s:element name="GetSensorListResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="GetSensorListResult"
type="s:string" />
```

```
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="SendCommand">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="clientName" type="s:string"
/>
            <s:element minOccurs="0" maxOccurs="1" name="sensorSite" type="s:string" />
            <s:element minOccurs="0" maxOccurs="1" name="command" type="s:string" />
            <s:element minOccurs="0" maxOccurs="1" name="args"
type="tns:ArrayOfAnyType" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="ArrayOfAnyType">
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="unbounded" name="anyType"
nillable="true" />
        </s:sequence>
      </s:complexType>
      <s:element name="SendCommandResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="SendCommandResult"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
  <wsdl:message name="RegisterControlSoapIn">
    <wsdl:part name="parameters" element="tns:RegisterControl" />
  </wsdl:message>
  <wsdl:message name="RegisterControlSoapOut">
    <wsdl:part name="parameters" element="tns:RegisterControlResponse" />
  </wsdl:message>
  <wsdl:message name="UnregisterControlSoapIn">
    <wsdl:part name="parameters" element="tns:UnregisterControl" />
  </wsdl:message>
  <wsdl:message name="UnregisterControlSoapOut">
    <wsdl:part name="parameters" element="tns:UnregisterControlResponse" />
  </wsdl:message>
  <wsdl:message name="GetSensorInfoSoapIn">
    <wsdl:part name="parameters" element="tns:GetSensorInfo" />
```

```
</wsdl:message>
<wsdl:message name="GetSensorInfoSoapOut">
  <wsdl:part name="parameters" element="tns:GetSensorInfoResponse" />
</wsdl:message>
<wsdl:message name="GetSensorInfoByNameSoapIn">
  <wsdl:part name="parameters" element="tns:GetSensorInfoByName" />
</wsdl:message>
<wsdl:message name="GetSensorInfoByNameSoapOut">
  <wsdl:part name="parameters" element="tns:GetSensorInfoByNameResponse" />
</wsdl:message>
<wsdl:message name="GetSensorListSoapIn">
  <wsdl:part name="parameters" element="tns:GetSensorList" />
</wsdl:message>
<wsdl:message name="GetSensorListSoapOut">
  <wsdl:part name="parameters" element="tns:GetSensorListResponse" />
</wsdl:message>
<wsdl:message name="SendCommandSoapIn">
  <wsdl:part name="parameters" element="tns:SendCommand" />
</wsdl:message>
<wsdl:message name="SendCommandSoapOut">
  <wsdl:part name="parameters" element="tns:SendCommandResponse" />
</wsdl:message>
<wsdl:portType name="SensorControlWSSoap">
  <wsdl:operation name="RegisterControl">
    <wsdl:input message="tns:RegisterControlSoapIn" />
    <wsdl:output message="tns:RegisterControlSoapOut" />
  </wsdl:operation>
  <wsdl:operation name="UnregisterControl">
    <wsdl:input message="tns:UnregisterControlSoapIn" />
    <wsdl:output message="tns:UnregisterControlSoapOut" />
  </wsdl:operation>
  <wsdl:operation name="GetSensorInfo">
    <wsdl:input message="tns:GetSensorInfoSoapIn" />
    <wsdl:output message="tns:GetSensorInfoSoapOut" />
  </wsdl:operation>
  <wsdl:operation name="GetSensorInfoByName">
    <wsdl:input message="tns:GetSensorInfoByNameSoapIn" />
    <wsdl:output message="tns:GetSensorInfoByNameSoapOut" />
  </wsdl:operation>
  <wsdl:operation name="GetSensorList">
    <wsdl:input message="tns:GetSensorListSoapIn" />
    <wsdl:output message="tns:GetSensorListSoapOut" />
  </wsdl:operation>
  <wsdl:operation name="SendCommand">
    <wsdl:input message="tns:SendCommandSoapIn" />
```

```
      <wsdl:output message="tns:SendCommandSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="SensorControlWSSoap" type="tns:SensorControlWSSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="RegisterControl">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/RegisterControl"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="UnregisterControl">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/UnregisterControl"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetSensorInfo">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetSensorInfo"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetSensorInfoByName">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetSensorInfoByName"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetSensorList">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetSensorList"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="SendCommand">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/SendCommand"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:binding name="SensorControlWSSoap12" type="tns:SensorControlWSSoap">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="RegisterControl">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/RegisterControl"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="UnregisterControl">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/UnregisterControl"
style="document" />
      <wsdl:input>
```

```
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetSensorInfo">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetSensorInfo"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetSensorInfoByName">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetSensorInfoByName"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="GetSensorList">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/GetSensorList"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="SendCommand">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/SendCommand"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="SensorControlWS">
    <wsdl:port name="SensorControlWSSoap" binding="tns:SensorControlWSSoap">
      <soap:address
location="http://localhost/SPAWARWebServices/SensorControlWS/SensorControlWS.a
smx" />
    </wsdl:port>
    <wsdl:port                                        name="SensorControlWSSoap12"
binding="tns:SensorControlWSSoap12">
      <soap12:address
location="http://localhost/SPAWARWebServices/SensorControlWS/SensorControlWS.a
smx" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

C. SENSORDATAPUBLISHERWS

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions                    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tns="http://Spawar.navy.mil/Code2644/WebServices/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
targetNamespace="http://Spawar.navy.mil/Code2644/WebServices/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <s:schema                                        elementFormDefault="qualified"
targetNamespace="http://Spawar.navy.mil/Code2644/WebServices/">
      <s:element name="PublishSensorData">
        <s:complexType>
          <s:sequence>
            <s:element  minOccurs="0"  maxOccurs="1"  name="clientName"  type="s:string"
/>
            <s:element      minOccurs="0"      maxOccurs="1"      name="sensorMessages"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="PublishSensorDataResponse">
        <s:complexType>
          <s:sequence>
            <s:element   minOccurs="0"   maxOccurs="1"   name="PublishSensorDataResult"
type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </wsdl:types>
  <wsdl:message name="PublishSensorDataSoapIn">
    <wsdl:part name="parameters" element="tns:PublishSensorData" />
  </wsdl:message>
  <wsdl:message name="PublishSensorDataSoapOut">
    <wsdl:part name="parameters" element="tns:PublishSensorDataResponse" />
  </wsdl:message>
  <wsdl:portType name="SensorDataPublisherWSSoap">
    <wsdl:operation name="PublishSensorData">
      <wsdl:input message="tns:PublishSensorDataSoapIn" />
      <wsdl:output message="tns:PublishSensorDataSoapOut" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding                                          name="SensorDataPublisherWSSoap"
type="tns:SensorDataPublisherWSSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="RegisterForPublishingSensorData">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/RegisterForPublishingSens
orData" style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="PublishSensorData">
      <soap:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/PublishSensorData"
style="document" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:binding                                    name="SensorDataPublisherWSSoap12"
type="tns:SensorDataPublisherWSSoap">
    <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="RegisterForPublishingSensorData">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/RegisterForPublishingSens
orData" style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
    <wsdl:operation name="PublishSensorData">
      <soap12:operation
soapAction="http://Spawar.navy.mil/Code2644/WebServices/PublishSensorData"
style="document" />
      <wsdl:input>
        <soap12:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap12:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="SensorDataPublisherWS">
    <wsdl:port                                      name="SensorDataPublisherWSSoap"
binding="tns:SensorDataPublisherWSSoap">
      <soap:address
location="http://localhost/SPAWARWebServices/SensorDataPublisherWS/SensorDataP
ublisherWS.asmx" />
    </wsdl:port>
    <wsdl:port                                    name="SensorDataPublisherWSSoap12"
binding="tns:SensorDataPublisherWSSoap12">
      <soap12:address
location="http://localhost/SPAWARWebServices/SensorDataPublisherWS/SensorDataP
ublisherWS.asmx" />
    </wsdl:port>
  </wsdl:service>
```

```
</wsdl:definitions>
```

LIST OF REFERENCES

[1] USN Program Executive Office for Command, Control, Communications, Computers and Intelligence (PEO C4I), “Net-Centric Implementation Framework, Part 1: Overview, Version 2.1.0,” 12 October 2007.

[2] Erl, Thomas, *Service-Oriented Architecture: Concepts, Technology, and Design*, Prentice Hall, Boston, MA 02116 2005.

[3] Thesis, “Service oriented architecture for coast guard command and control” by Russell E. Dash and Robert H. Creigh, March 2007.

[4] O’Brien, Liam, Len Bass, and Paulo Merson. “Quality Attributes and Service-Oriented Architectures.” *Software Engineering Institute Technical Note*. CMU/SEI-2005-TN-014. Sep. 2005.
URL*: http://www.sei.cmu.edu/publications/documents/05.reports/05tn014.html*

[5] NESI Part 1, v1.3, 16 June 2006.

[6] OASIS: Advanced Open Standards for the Information Society. *UDDI Version 2.04 API Specification UDDI Committee Specification*. 2002.
URL: *http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf*

[7] W3CSchools.com. *XML Tutorial*. URL:
http://www.w3schools.com/xml/default.asp

[8] W3C. *Web Services Description Language (WSDL)*. 2001. URL:
http://www.w3.org/TR/wsdl

[9] W3Schools.com. *SOAP Tutorial*. URL:
http://www.w3schools.com/soap/default.asp

[10] OASIS: Advanced Open Standards for the Information Society. *Web Services Coordination (WS-Coordination) Version 1.2*. 2009. URL:
http://docs.oasis-open.org/ws-tx/wstx-wscoor-1.2-spec-os/wstx-wscoor-1.2-spec-os.html

[11] W3C. *Web Services Addressing (WS-Addressing)*. 2004. URL:
http://www.w3.org/Submission/ws-addressing/

[12] IBM. *Web Services Notification (WS-Notification*)
2004. URL: *http://www.ibm.com/developerworks/library/ws-resource/ws-notification.pdf*

[13] W3C. *Web Services Eventing (WS-Eventing).* 2006. URL: *http://www.w3.org/Submission/WS-Eventing/*

[14] The Internet Engineering Task Force (IETF). *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: *http://www.ietf.org/rfc/rfc2616.txt*

[15] The Internet Engineering Task Force (IETF). *Simple Mail Transfer Protocol*. 1982. URL: *http://tools.ietf.org/html/rfc821*

[16] The Internet Engineering Task Force (IETF). *Transfer Control Protocol (FTP).* 1985. URL: *http://tools.ietf.org/html/rfc959*

[17] OASIS: Advanced Open Standards for the Information Society. *Web Services Security: SOAP Message Security 1.0 (WS-Security)* 2004. URL: *http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf*

[18] xmlsoap.org. *Web Services Security Policy Language (WS-SecurityPolicy).* 2005. URL: *http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf*

[19] xmlsoap.org. *Web Services Secure Conversation Language (WS-SecureConversation).* 2005. URL: *http://specs.xmlsoap.org/ws/2005/02/sc/WS-SecureConversation.pdf*

[20] xmlsoap.org. *Web Services Trust Language (WS Trust)*. 2005. URL: *http://specs.xmlsoap.org/ws/2005/02/trust/WS-Trust.pdf*

[21] IBM. *Web Service Federation Language*. 2003. URL: *http://www.ibm.com/developerworks/library/specification/ws-fed/*

[22] Online Community of the Security Assertion Markup Language (SAML) OASIS Standard. *SAML Specification*. 2008. URL: *http://saml.xml.org/saml-specifications*

[23] IBM. *Web Service Manageability*. 2003. URL: *http://www.ibm.com/developerworks/library/specification/ws-manage/*

[24] Object Management Group. *Documents Associated with Business Process Model and Notation (BPMN) 1.2.* Jan. 2009. URL: *http://www.omg.org/spec/BPMN/1.2/*

[25] .NET Framework Developer Center. *.NET Remoting Overview*. 2003. URL: *http://msdn.microsoft.com/en-us/library/kwdt6w2k(VS.71).aspx*

[26] W3C. *XML-binary Optimized Packaging*. Jan. 2005. URL: *http://www.w3.org/TR/xop10/*

[27] W3C. *SOAP Message Transmission Optimization Mechanism*. Jan. 2005. URL: *http://www.w3.org/TR/soap12-mtom/*

[28] IBM. *WebSphere DataPower SOA Appliances*. 2009. URL: *http://www-01.ibm.com/software/integration/datapower/*

[29] IBM. *WebSphere Application Server*. 2009. URL: *http://www-01.ibm.com/software/webservers/appserv/was/*

[30] OASIS: Advanced Open Standards for the Information Society. *eXtensible Access Control Markup Language (XACML) TC*. 2008. URL: *http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml*

[31] OASIS*: Web Services Reliable Messaging (WS-ReliableMessaging) . 2009. URL: http://docs.oasis-open.org/ws-rx/wsrm/200702*

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. SPAWAR 05
   Space and Naval Warfare Systems Center
   San Diego, CA

4. Program Executive Officer Command, Control, Communications, Computers, and Intelligence
   Space and Naval Warfare Systems Center
   San Diego, CA

5. Man-Tak Shing
   Naval Postgraduate School
   Monterey, CA

6. James Bret Michael
   Naval Postgraduate School
   Monterey, CA